4. Speicher- und Prozeßverwaltung

J. Dunkel

4.1 Einleitung

In diesem Kapitel soll gezeigt werden, wie durch die Architektur moderner Mikroprozessoren, insbesondere von Seiten der Hardware, die Grundlagen für die Implementierung von Betriebssystemen (*operating systems*) geschaffen werden. Dabei kann dieses Kapitel natürlich nicht ein ganzes Buch zum Thema "Betriebssysteme" ersetzen. Zur Klärung detaillierterer Fragen bzgl. der Struktur und der Implementierung von Betriebssystemen sei deshalb auf andere Bücher über Betriebssysteme, Leistungsbewertung von Rechensystemen usw. verwiesen. Ebenso soll hier nicht die Benutzeroberfläche bestimmter Mikrorechner-Betriebssysteme, wie beispielsweise MS-DOS oder UNIX vorgestellt werden.

In diesem Buch aus dem Bereich der technischen Informatik geht es darum zu zeigen, wie von Seiten der Hardware die von einem Betriebssystem zu leistenden Aufgaben unterstützt werden. Wir betrachten dabei speziell nur Betriebssysteme für Mikrorechner-Systeme, wobei insbesondere die Konzepte und Möglichkeiten der modernen 16/32-bit-Prozessoren vorgestellt werden. Vorausgesetzt wird dabei stets, daß der Rechner nur einen zentralen Prozessor besitzt. Nicht betrachtet werden also Mehrprozessor-Systeme mit ihren speziellen (verteilten) Betriebssystemen.

Zunächst wollen wir eine kleine Auswahl der Definitionen des Begriffs "Betriebssystem" aus der Literatur wiedergeben. Die unterschiedlichen Beschreibungen resultieren aus der Vielfalt der Aufgaben, die ein Betriebssystem erfüllen muß. In den folgenden Unterabschnitten werden wir ausführlicher die allgemeinen Aufgaben und Problemkreise von Betriebssystemen beschreiben.

4.1.1 Definitionen des Begriffs "Betriebssystem"

Das Deutsche Institut für Normung definiert (DIN 44300):

> "Ein Betriebssystem umfaßt die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen."

Die amerikanische Norm ANS (*American National Standard*) legt fest:

"Software, which controls the execution of computer programs and which may provide scheduling, debugging, input/output control, accounting, compilation, storage assignment, data management and related services."

In A.N. Haberman: "Entwurf von Betriebssystemen", Springer 1981, wird definiert:

"Ein Betriebssystem setzt sich aus Programmen zusammen, welche die Ausführung von Benutzerprogrammen und die Benutzung von Betriebsmitteln überwachen."

4.1.2 Ziele von Betriebssystemen

Wie aus den vorhergehenden und den folgenden Kapiteln ersichtlich wird, ist ein Mikrorechner ein komplexes System aus Hardware-Bausteinen, deren Benutzung sehr detaillierter Kenntnisse bedarf. Den Benutzer, der bestimmte Anwendungen mit dem Rechner bearbeiten will (z.B. Textverarbeitung, Ausführung eines PASCAL-Programms, usw.), interessiert aber nicht, wie seine Anwendungen durch die Hardware gelöst werden, sondern nur, daß sie – möglichst schnell und zuverlässig – bearbeitet werden können. Er wünscht sich eine Maschine, die beispielsweise Texte verarbeiten oder PASCAL-Programme ausführen kann. Offensichtlich besteht eine Diskrepanz zwischen den Fähigkeiten und Diensten, die Hardware-Bausteine zur Verfügung stellen und den vom Benutzer erwünschten Fähigkeiten des Rechensystems. Deshalb spricht man hier von der semantischen Lücke (vgl. Bild 4.1-1).

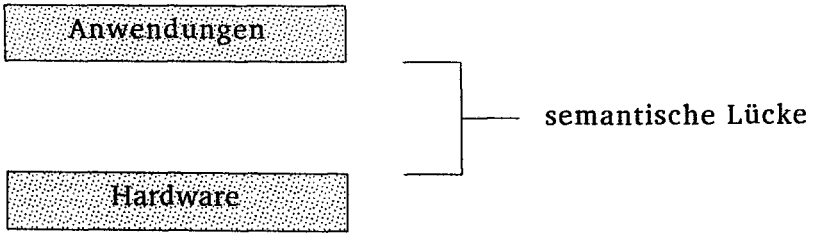


Bild 4.1-1. Zum Begriff der semantischen Lücke

Die semantische Lücke wird – wie im Bild 4.1-2 dargestellt – teilweise vom Betriebssystem ausgefüllt, das für den Benutzer und den Betreiber eines Rechensystems die Fähigkeiten der Hardware erweitert. Dabei stehen drei Gesichtspunkte im Vordergrund:

- Benutzerfreundlichkeit,
- Effizienz und Zuverlässigkeit,
- Portabilität.

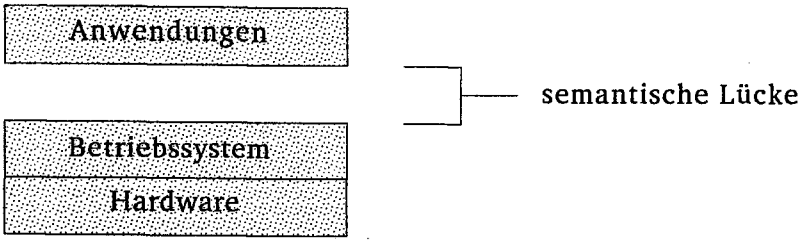


Bild 4.1-2. Das Betriebssystem verkleinert die semantische Lücke

4.1.2.1 Benutzerfreundlichkeit

Der Benutzer des Rechensystems soll sich in erster Linie um seine Anwendungen kümmern können. Für ihn irrelevante Details der Hardware sollten ihm verborgen bleiben (*information hiding*), ohne daß seine Anwendungswünsche eingeschränkt werden. Deshalb stellt ihm das Betriebssystem komplexe Dienste für seine Aufgaben zur Verfügung, deren Benutzung aber keine genauen Kenntnisse der Hardware erfordert.

Beispiel

Der Benutzer soll Dateien im Hauptspeicher (Arbeitsspeicher) ablegen können, ohne sich selber darum kümmern zu müssen, ob und wo noch genügend freier Speicherplatz vorhanden ist. Insbesondere muß er selbst nicht die physikalischen Speicheradressen bestimmen. Solche Verwaltungsaufgaben werden ihm vollständig vom Betriebssystem abgenommen.

Das Betriebssystem stellt somit eine "Maschine" zur Verfügung, die komplexere Funktionen übernehmen kann als die Hardware des Rechners für sich allein. Man spricht von einer virtuellen Maschine, denn der Benutzer kann nicht trennen, welche Aufgaben direkt von der Hardware und welche vom Betriebssystem übernommen werden. Der Rechner besitzt für ihn vielmehr die Fähigkeiten und Eigenschaften, die vom Betriebssystem zur Verfügung gestellt werden. Das Betriebssystem bildet somit die Schnittstelle des Benutzers zur Hardware.

4.1.2.2 Effizienz und Zuverlässigkeit

Den Betreiber eines Rechensystems interessiert aber nicht nur eine einfache, ohne spezielle Systemkenntnisse mögliche Benutzung, sondern er wünscht sich zusätzlich eine gute Auslastung seiner Anlage. D.h. die einzelnen Komponenten des Rechensystems – wie z.B. der Prozessor – sollen möglichst durchgehend mit Arbeit beschäftigt sein. Für eine effiziente Nutzung des Rechensystems wird somit eine möglichst intensive, parallele Nutzung aller Hardware-Komponenten gefordert. Insbesondere sollen die Zeiten, in denen der Prozessor unbeschäftigt (*idle*) ist, möglichst klein sein. Bei Großrechnern versucht man schon seit langem, Effizienz

dadurch zu erreichen, daß nicht nur ein einziger Benutzer mit dem Rechner arbeitet, sondern daß mehrere Benutzer gleichzeitig Zugang zum Rechner erhalten.

Dazu werden mehrere Benutzerprogramme in den Hauptspeicher geladen, die abwechselnd von der CPU bearbeitet werden. Bei jedem Rechensystem ist die Peripherie, z.B. Drucker, Plattenspeicher usw., um Größenordnungen langsamer als der Prozessor selbst. Sobald ein Benutzerprogramm eine Ein-/Ausgabeoperation durchführen muß und der Prozessor somit unbeschäftigt ist, weist das Betriebssystem dem frei gewordenen Prozessor ein anderes Programm zur Bearbeitung zu. Betriebssysteme, die diese Betriebsart unterstützen, werden Mehrprogramm-Betriebssysteme (*multiprogramming*) genannt. Nachteilig bei dieser Betriebsart eines Rechners ist, daß auch wichtige Aufgaben u.U. lange auf die Zuteilung des Prozessors warten müssen. Dieser Nachteil wird durch die im folgenden beschriebene Betriebsart vermieden.

Insbesondere für bestimmte Aufgaben bei der Steuerung von Fertigungs-Prozessen ist es unbedingt erforderlich, daß vom Rechensystem mehrere Aufgaben (nahezu) parallel bearbeitet werden können. Dazu werden mehrere Aufträge gleichzeitig in den Hauptspeicher eingelagert und alternierend vom Prozessor bearbeitet. Dies wird dadurch realisiert, daß die CPU sehr schnell zwischen der Bearbeitung verschiedener Aufgaben bzw. Programmen wechselt, jeder Auftrag also nur für einen winzig kleinen Zeitraum im Millisekunden-Bereich ("Zeitscheibe" – *time slice*) den Prozessor zugeteilt bekommt und dann mit der Bearbeitung eines anderen Auftrags fortgefahren wird. Der Benutzer gewinnt dadurch den Eindruck, daß mehrere Prozesse wirklich parallel vom Prozessor ausgeführt werden.

Ein in Ausführung befindliches oder ausführbereites Programm, zusammen mit seinen Daten, also den Variablen und Konstanten mit ihren aktuellen Werten, wird ein **Prozeß** genannt oder synonym mit dem englischen Begriff *Task* bezeichnet. Betriebssysteme, die die eben beschriebene Aufgabenbearbeitung (*process multiplexing*) ermöglichen, heißen deshalb *Multitasking*-Betriebssysteme. Ein Beispiel für ein solches Betriebssystem im Mikrorechner-Bereich ist UNIX.

Die Möglichkeit des Multitaskings, die in der Vergangenheit nur bei Großrechnern gegeben war, steht inzwischen auch Mikroprozessor-Systemen zur Verfügung. In diesem Kapitel wird gezeigt, wie dieses Konzept durch bestimmte Hardware-Eigenschaften unterstützt wird.

Zum Wunsch nach Effizienz tritt noch die Forderung nach Zuverlässigkeit. Gerade wenn sich mehrere Benutzer bzw. Prozesse ein Rechensystem teilen, entstehen eine Reihe von Problemen. So muß gesichert sein, daß ein Prozeß nicht durch andere gestört werden kann. Eine Störung könnte z.B. darin bestehen, daß wichtige Daten von anderen Prozessen überschrieben werden. Um solche Fehler zu verhindern, müssen von Seiten des Betriebssystems und der Hardware eine Reihe von Schutzmechanismen (*protections*) bereit gestellt werden.

Bei sehr vielen Anwendungen ist ihre fehlerfreie Ausführung von entscheidender Bedeutung. Ein Betriebssystem muß deshalb gewährleisten, daß auftretende Fehler und Ausnahmesituationen möglichst abgefangen werden und nicht zu einem Systemzusammenbruch führen (*exception handling*). Wie dies geschieht, wurde bereits im Abschnitt 1.12 angesprochen.

4.1.2.3 Portabilität

Eine weitere wichtige Anforderung ist, daß auf einem bestimmten Rechner entwickelte Software auch auf andere Rechensysteme übertragen werden kann. Insbesondere will man oft benutzte Standard-Software (wie Compiler oder Textverarbeitungssysteme) auf möglichst vielen Rechnern ohne aufwendige Anpassungen benutzen können. Wie bereits erwähnt, stellt das Betriebssystem eine allgemeine und abstrakte Schnittstelle zur Hardware dar und erleichtert so die Portabilität von Software. So bildet z.B. das Betriebssystem MS-DOS die Schnittstelle für sehr viele Anwendungsprogramme im Bereich der Personal Computer. Für die Ablauffähigkeit des Anwendungsprogramms kann bei gleichem Betriebssystem die zugrundeliegende Hardware (z.B. der Prozessor) durchaus unterschiedlich sein.

4.1.3 Spezielle Aufgaben von Betriebssystemen

Im folgenden sollen die vom Betriebssystem zu lösenden Aufgaben und Problemkreise kurz vorgestellt werden. In Analogie zu den meisten Lehrbüchern unterscheiden wir die folgenden Hauptaufgaben:

- Auftragsverwaltung,
- Speicherverwaltung,
- Betriebsmittelverwaltung.

4.1.3.1 Auftragsverwaltung (*task management*)

Eine der wichtigsten Aufgabe von Betriebssystemen ist die Verwaltung von Aufträgen, also der oben definierten Prozesse (Tasks). Ein Prozeß kann im Laufe seiner Bearbeitung die folgenden, unterscheidbaren Zustände annehmen:

aktiv (*running*) Der Prozeß wird gerade vom Prozessor bearbeitet;

blockiert (*suspended*) Der Prozeß muß auf ein bestimmtes Ereignis warten, z.B.
- auf das Ende einer Ein-/Ausgabeoperation,
- auf die Einlagerung von Daten in den Hauptspeicher,
- auf einen anderen Prozeß;

bereit (*ready*) Der Prozeß ist bereit, vom Prozessor ausgeführt zu werden, und muß insbesondere auf kein Ereignis warten.

Wie bereits erwähnt, ist es bei vielen Anwendungen, insbesondere bei Multitasking-Betriebssystemen, wünschenswert, daß mehrere Aufgaben quasi gleichzeitig abgearbeitet werden. Das Betriebssystem muß dazu eine spezielle Routine, den sogenannten *Dispatcher*, zur Verfügung stellen, der das Umschalten der CPU auf verschiedene Aufträge bewerkstelligt. Die typische Arbeitsweise eines *Dispatchers* ist die folgende:

Jedesmal wenn ein Prozeß in den Zustand "blockiert" übergeht, kann die CPU von diesem nicht mehr genutzt werden. In dieser Situation kann der Dispatcher der CPU einen anderen "bereiten" Prozeß zuteilen. Dadurch bleibt der Prozessor

beschäftigt (*busy*), und das System wird effizienter genutzt. Sobald das Ereignis eingetroffen ist, das einen Prozeß in den Zustand "blockiert" versetzt hat, kann er gegebenenfalls wieder vom Prozessor ausgeführt werden, d.h. er geht in den Zustand "bereit" über. Diese Zustandsübergänge werden im Bild 4.1-3 dargestellt.

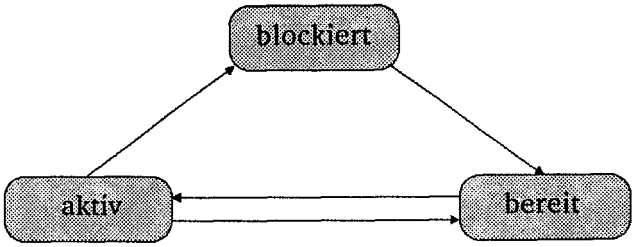


Bild 4.1-3. Übergänge zwischen den verschiedenen Prozeß-Zuständen.

4.1.3.2 Speicherverwaltung (*memory management*)

Durch immer größer werdende Programme und die Möglichkeit von Multitasking-Betriebssystemen, mehrere Aufträge quasi gleichzeitig, verzahnt von der CPU ausführen zu lassen, werden die Grenzen des vorhandenen Hauptspeichers schnell erreicht. Um trotz dieses Hauptspeicher-Engpasses die genannten Aufgaben bewältigen zu können, wurde das Verfahren der virtuellen Speicherverwaltung entwickelt.

Virtuelle Speicherverwaltungssysteme nutzen die Lokalitätseigenschaft von Programmen aus. Darunter versteht man die Beobachtung, daß Prozesse während ihrer Ausführung im allgemeinen nicht ständig sämtliche Daten und den gesamten Programmcode benutzen, sondern in jedem Zeitintervall jeweils nur einen kleinen Teil davon. Aus diesem Grund ist es ausreichend, nur den momentan benötigten Anteil an Programmcode und Daten, die sogenannte Arbeitsmenge (*working set*), in den Hauptspeicher einzulagern. Die restlichen Teile des Programms sowie die nicht benötigten Daten befinden sich im Hintergrundspeicher (Peripheriespeicher). Benötigt ein Prozeß Programmteile oder Daten, die sich nicht im Hauptspeicher befinden, so müssen sie vom Hintergrundspeicher in den Hauptspeicher eingelagert werden (*swapping, paging*).

Der Benutzer selbst merkt nichts von dieser aufwendigen Speicherverwaltung. Insbesondere kann er nicht erkennen, daß ihm nur ein begrenzter Hauptspeicherplatz zur Verfügung steht. Der Vorgang des häufigen Austauschens von Programmen und Daten zwischen Hintergrund- und Hauptspeicher bleibt dem Benutzer verborgen, d.h. er ist ihm vollständig transparent. Weil offensichtlich nur die gerade benötigte Arbeitsmenge eingelagert wird, kann ein Programm und seine Daten durchaus einen Speicherbedarf besitzen, der die Hauptspeichergröße bei weitem übersteigt. Ein nach diesem Konzept verwalteter Speicher wird des-

halb auch **virtueller Speicher** genannt (s. Bild 4.1-4). (Beachten Sie bitte schon hier, daß Programme und ihre Daten sowohl im Hintergrund- wie im Hauptspeicher nicht zusammenhängende Speicherbereiche belegen, sondern fast beliebig über diese verteilt sein können.)

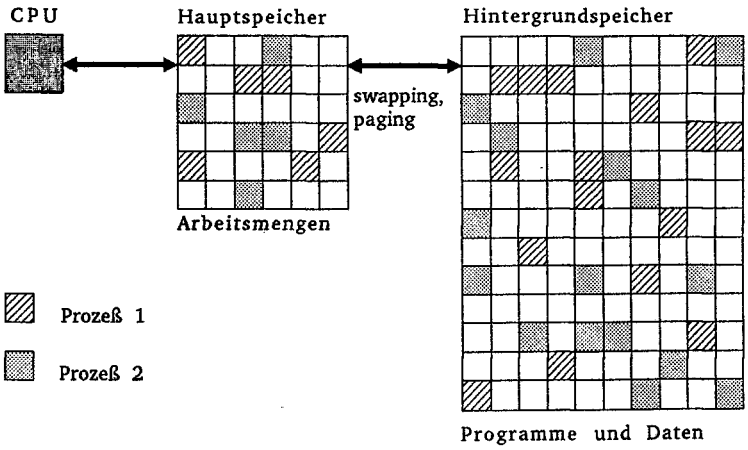


Bild 4.1-4. Grundstruktur virtueller Speicherverwaltung

Eine effizient arbeitende virtuelle Speicherverwaltung läßt sich dann erreichen, wenn sie möglichst weitgehend durch die Hardware unterstützt wird. Bei den 16- bzw. 32-bit-Prozessoren der neuesten Generation ist dies fast ausnahmslos der Fall. Im Mikroprozessor-Chip ist dann bereits eine Speicherverwaltungseinheit (*memory management unit* – MMU) integriert, oder sie ist in einem speziellen Baustein realisiert. Die Architektur und Arbeitsweise dieser MMU's vorzustellen, ist ein Anliegen dieses Kapitels.

4.1.3.3 Betriebsmittelverwaltung (*resource management*)

Neben den oben genannten zentralen Aufgaben des Betriebssystems müssen auch alle restlichen Betriebsmittel (z.B. Festplatte, Floppy Disk, Drucker, sowie sonstige E/A-Geräte) verwaltet werden. Auf die Betriebsmittelverwaltung gehen wir im Rahmen dieses Buches nur sehr knapp ein. Im Kapitel 6 stellen wir dazu die Grundlagen der Speicherplatz-Verwaltung in einem Festplatten- oder Floppy-Disk-System dar.

Überblick

In den folgenden Abschnitten dieses Kapitels sollen die Hardware-Konzepte moderner Mikroprozessoren und ihrer Peripheriebausteine vorgestellt werden, die eine Implementierung von Multitasking-Betriebssystemen unterstützen. Dazu

stellen wir zunächst die Grundkonzepte der virtuellen Speicherverwaltung vor. Insbesondere wird die Aufteilung des Speichers in Segmente unterschiedlicher Länge einer Aufteilung in Seiten fester Länge gegenübergestellt. Nach diesen allgemeineren Ausführungen wird die virtuelle Speicherverwaltung bei konkreten Mikroprozessoren vorgestellt.

Es folgen Beschreibungen der bei Multitasking-Betriebssystemen erforderlichen Schutzmechanismen sowie der Mittel, mit denen die Hardware den Wechsel zwischen verschiedenen Prozessen unterstützt.

Oftmals ist es erforderlich, daß in einem System mehrere Prozesse miteinander kommunizieren können. Die zur Kommunikation zwischen Prozessen benötigten Mechanismen sind Gegenstand eines eigenen Abschnitts. Eine weitere wichtige Aufgabe von Betriebssystemen ist die Verwaltung von *Interrupts* und *Exceptions*. Wie diese durch die Hardware unterstützt wird, beschreiben wir zum Schluß dieses Kapitels.

4.2 Einführung in die Speicherverwaltung

In diesem Abschnitt werden die allgemeinen Grundlagen der virtuellen Speicherverwaltung vorgestellt und insbesondere die beiden Ansätze einer Segment- und einer Seiten-orientierten Speicherverwaltung gegenübergestellt.

4.2.1 Virtuelle Speicher

Neue Anwendungen im Mikrorechnerbereich, wie z.B. der Einsatz von Datenbanken, CAD-Systemen oder hochauflösender Graphik, führen zu einem ständigen Anwachsen der Anforderungen an Speicherplatz. Auch durch den Einsatz von Multitasking-Betriebssystemen im PC-Bereich (z.B. UNIX) mit der Möglichkeit, mehrere Aufträge quasi gleichzeitig von der CPU bearbeiten zu lassen, werden die Grenzen des vorhandenen Hauptspeichers schnell erreicht. Sogar bei rasch sinkenden Preisen für Halbleiterspeicher-Bausteine kann der Hauptspeicher nicht im Tempo der steigenden Anforderungen vergrößert werden.

Um trotz dieses Hauptspeicher-Engpasses eine Multitasking-Betriebsart realisieren zu können, wurden bei Großrechnern schon frühzeitig Verfahren zur virtuellen Speicherverwaltung entwickelt. Dabei befinden sich nur Teile des Programms sowie der benötigten Daten im Hauptspeicher, während Programmcode und Daten in ihrer Gesamtheit im Hintergrundspeicher (z.B. auf einer Festplatte) residieren. Werden Daten benötigt, die sich aktuell nicht im Hauptspeicher befinden, so müssen sie bei Bedarf vom Hintergrundspeicher in den Hauptspeicher eingelagert werden (*swapping*). Der Austausch von Daten zwischen Hintergrund- und Hauptspeicher kostet natürlich Zeit. Trotzdem stellt dieses Verfahren einen Kompromiß zwischen Speicherkosten und Zugriffsgeschwindigkeit dar, weil bei einer geschickten Speicherverwaltung die Zugriffe auf den Hintergrundspeicher recht selten erforderlich sind.

Als Rechtfertigung für ein solches Vorgehen wurde bereits die Lokalitätseigenschaft von Prozessen genannt: Programme benutzen während ihrer Ausführung im allgemeinen nicht ständig sämtliche Daten und den gesamten Programmcode, sondern nur einen kleinen Teil davon. Meist werden bestimmte Programmschleifen, Prozeduren und Datenstrukturen wiederholt vom Programm benutzt, bevor andere Programmteile ausgeführt werden. Durch die modulare Programmentwicklung moderner Programmiersprachen wird diese Eigenschaft noch verstärkt. Aus diesem Grund ist es ausreichend, nur die momentan benötigte Untermenge an Code und Daten in den Hauptspeicher einzulagern, die sogenannte Arbeitsmenge (*working set*). In der Literatur über Betriebssysteme werden eine Vielzahl von Verfahren untersucht, um eine geeignete Arbeitsmenge für jedes Programm zu bestimmen. Die Wahl der Größe der Arbeitsmenge legt fest, wie oft Daten aus dem Hintergrundspeicher in den Hauptspeicher eingelagert werden müssen. Sie ist von entscheidender Bedeutung für die Systemleistung:

Wird die Arbeitsmenge zu klein gewählt, dann müssen bei der Auftragsausführung fast ständig Daten vom Hintergrundspeicher eingelagert werden, so daß im Extremfall das System fast nur noch mit dem Einlagern beschäftigt ist. Dieser Effekt wird in der englischsprachigen Literatur mit *Thrashing* (Seitenflattern) bezeichnet. Wird die Arbeitsmenge zu groß gewählt, dann sind für das Programm mehr Daten im Hauptspeicher eingelagert, als eigentlich gerade benötigt werden. Das Rechensystem wird in diesem Fall nicht effizient genutzt, da es wegen Speicherplatzmangels weniger Aufgaben als eigentlich möglich parallel abarbeiten kann.

Die Hauptaufgabe der virtuellen Speicherverwaltung ist die Umsetzung virtueller (logischer) Adressen in physikalische Adressen (s. Bild 4.2-1).

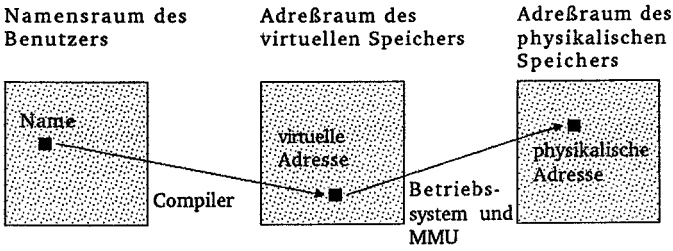


Bild 4.2-1. Bestimmung physikalischer Adressen

Die vom Benutzer in einer höheren Programmiersprache eingesetzten Namen der benutzten Objekte (Programme, Unterprogramme, Variablen, etc.) werden vom Compiler in **virtuelle Adressen** (logische Adressen) übersetzt, die noch nicht den Ort des gesuchten Objektes im Hauptspeicher bezeichnen. Dieser läßt sich vielmehr durch das Betriebssystem erst während der Ausführungszeit des Auftrags bestimmen, wenn die Stelle des Hauptspeichers feststeht, an der die benötigten

Programme oder Daten eingelagert werden. Sobald dies geschehen ist, kann die virtuelle Adresse auf eine **physikalische Adresse** abgebildet werden, d.h. die Adresse, mit der schließlich auf das gesuchte Speicherwort zugegriffen werden kann. Formal wird durch die MMU also die sogenannte Speicherabbildungs-Funktion

f: virtueller Adreßraum --> physikalischer Adreßraum

realisiert. Dabei ist normalerweise der physikalische Adreßraum sehr viel kleiner als der virtuelle.

Im folgenden soll noch genauer untersucht werden, wie bei verschiedenen Mikroprozessoren die Speicherabbildungs-Funktion realisiert wird. Zunächst unterscheiden wir jedoch zwei grundlegende Verfahren der virtuellen Speicherverwaltung, die Segmentierung und die Aufteilung in Seiten.

4.2.2 Segmentierungs- und Seitenwechsel-Verfahren

Segmentierung

Bei diesem Ansatz ist der virtuelle Adreßraum in **Segmente** verschiedener Länge unterteilt. Jedem Prozeß sind ein oder mehrere Segmente z.B. für den Programmcode und die Daten, zugeordnet. Die einzelnen Segmente enthalten logisch zusammenhängende Informationen und können relativ groß sein. Auf der Ebene höherer Programmiersprachen finden sie ihre Entsprechungen in den (Unter-) Programm- oder Datenmodule. Jeder Auftrag kommt im Normalfall mit einer relativ kleinen Anzahl von Segmenten aus. Die Festlegung der Segmente kann entweder durch den Benutzer oder durch den Compiler erfolgen.

Aufteilung in Seiten

Bei den Seitenwechselverfahren (*paging*) werden der logische und der physikalische Adreßraum in "Segmente fester Länge", die sogenannten **Seiten** (*pages*) unterteilt. (Physikalische Seiten im Hauptspeicher werden häufig auch als Rahmen (*frames*), seltener als Kacheln bezeichnet. Durch Seitenwechselverfahren verwaltete Speicher nennt man oft verkürzend Seitenspeicher oder seitenorientierte Speicher.) Auf Programmebene finden Speicherseiten meist keine direkte Entsprechung. Die Kapazität einer Seite ist relativ klein. Typisch sind bei Mikroprozessoren Seitengrößen zwischen 256 byte und 4 kbyte, so daß jeder Auftrag normalerweise eine Vielzahl von Seiten benötigt. Die Aufteilung des Adreßraums in Seiten ist für den Benutzer vollständig transparent, d.h. sie wird vom Betriebssystem mit entsprechender Hardware-Unterstützung durchgeführt, ohne daß der Benutzer davon etwas merkt und ohne daß er sich darum kümmern muß.

Im Mikroprozessorbereich gab es in den vergangenen Jahren heftige Kontroversen zwischen den Anhängern beider Verfahren. Intel und Motorola unterstützen mit ihren Typen 80286 und 68010 die Segmentierung, Zilog's Z8003/4 und National Semiconductor's 32000 arbeiten mit dem Seitenwechselverfahren. In den letzten Jahren deutet sich ein Kompromiß derart an, daß von einigen neueren 32-bit-

Prozessoren, wie z.B. dem Intel 80386 und 80486, beide Verfahren unterstützt werden.

Beide Ansätze haben in Abhängigkeit von der konkreten Anwendung durchaus ihre Berechtigung. Die Vor- und Nachteile, speziell bzgl. des Verwaltungsaufwandes und der Speicherplatzausnutzung, sollen im folgenden diskutiert werden.

4.2.2.1 Probleme der virtuellen Speicherverwaltung

Um den Austausch von Daten zwischen Haupt- und Hintergrundspeicher durchführen zu können, ergeben sich für das Betriebssystem drei verschiedene Problemkreise.

1. Der Einlagerungszeitpunkt

Das Betriebssystem muß festlegen, wann Daten in den Hauptspeicher eingelagert werden. Normalerweise geschieht dies sowohl bei Segmentierungs- als auch bei Seitenwechsel-Verfahren ausschließlich auf Anforderung, d.h. Daten werden dann eingelagert, wenn auf sie zugegriffen wird und sie sich nicht im Arbeitsspeicher befinden. Bei Seitenspeichern hat sich dafür in der Literatur der Begriff des *Demand Paging* durchgesetzt. Den Zugriff auf ein nicht im Hauptspeicher vorhandenes Segment bzw. eine Seite bezeichnet man auch als **Segment-** oder **Seiten-Fehler** (*segment fault, page fault*).

2. Das Zuweisungsproblem

Im Betriebssystem muß eine Strategie implementiert sein, die festlegt, an welche Stelle des Arbeitsspeichers nicht vorhandene Daten eingelagert werden.

Bei **Segmentierungsverfahren** stellt sich für jedes einzulagernde Segment das Problem, im Arbeitsspeicher eine ausreichend große Lücke, d.h. einen zusammenhängenden freien Speicherbereich, zu finden. Lücken entstehen, wenn nicht mehr benötigte Segmente ausgelagert werden. Diese Auslagerung wird z.B. dann durchgeführt, wenn eine Datei geschlossen wird oder ein Prozeß terminiert. Da die Segmente keine einheitliche Größe besitzen, wird für ein einzulagerndes Segment im allgemeinen keine Lücke gefunden, in die es ganz genau hineinpaßt. Vom Betriebssystem können dann unterschiedliche Strategien benutzt werden, um eine geeignete Lücke zu finden. Die bekanntesten Zuweisungsstrategien sind:

first-fit: Die Lücken sind nach aufsteigenden Anfangsadressen geordnet. Das Segment wird in die erste Lücke eingelagert, in die es hineinpaßt.

best-fit: Das Segment wird in die kleinste Lücke eingelagert, in die es gerade noch hineinpaßt.

worst-fit: Das Segment wird stets in die größte der zur Verfügung stehenden Lücken eingelagert.

Bei jeder der genannten Strategien zerfällt der Arbeitsspeicher nach einiger Zeit in belegte und unbelegte Speicherbereiche. Es stellt sich dann das Problem der externen Fragmentierung, d.h. es gibt viele Lücken, die so klein sind, daß kaum ein

Segment mehr hineinpaßt. Der zur Verfügung stehende Speicherplatz hat sich faktisch um diese Bereiche verkleinert. Im Beispiel, das Bild 4.2-2 zeigt, sind die belegten Speicherplätze schraffiert, die freien nicht-schraffiert dargestellt. Eine zunehmende externe Fragmentierung des Hauptspeichers läßt sich durch eine vom Betriebssystem in regelmäßigen Abständen durchgeführte Speicherverdichtung (Kompaktierung) beheben. D.h. alle im Speicher vorhandenen Daten werden so verschoben, daß ein zusammenhängender belegter und ein zusammenhängender freier Speicherbereich entstehen. Nach der Kompaktierung gibt es also keine Speicherfragmente mehr.

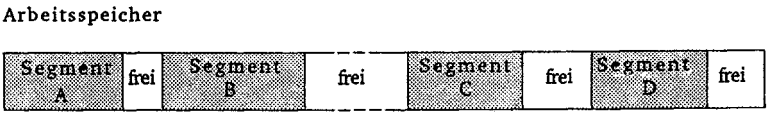
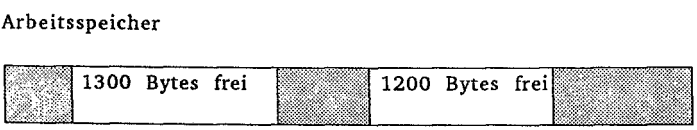


Bild 4.2-2. Externe Fragmentierung bei einem segmentierten Speicher

Selbsttestaufgabe S4.2-1:

Ein Arbeitsspeicher sei bis auf zwei Lücken der Größen 1300 und 1200 byte vollständig belegt.



Die nachfolgenden Speicheranforderungen benötigen Segmente der Größen 1000, 1100, 250 byte.

a) Wenden Sie die Zuweisungsalgorithmen *first-fit* und *best-fit* auf diese Folge von Speicheranforderungen an. Welcher der beiden Algorithmen ist günstiger ?

b) Vergleichen Sie beide Algorithmen bzgl. ihres Suchaufwandes und der Fragmentierung !

Bei **Seitenwechselverfahren** stellt sich das Zuweisungsproblem nicht, weil die Größe aller belegten und freien Speicherbereiche ganzzahlige Vielfache einer Seitengröße sind. Für jede einzulagernde Seite gibt es also immer eine genau pas-

sende Lücke, denn es werden stets "Segmente fester Länge" zwischen Hintergrund- und Arbeitsspeicher ausgetauscht. Aus diesem Grund tritt bei seitenorientierten Speichern das Problem der externen Fragmentierung nicht auf. Doch leider bedeutet das trotzdem keine vollständige Ausnutzung des Arbeitsspeichers. Bei Seitenspeichern tritt vielmehr das Problem der internen Fragmentierung auf. Denn die Datenmengen eines Auftrags entsprechen im allgemeinen nicht dem Vielfachen einer Seitengröße. Somit ist zumindest die "letzte" Seite eines Programms meist nicht vollständig genutzt. Bei modular-strukturierten Programmen verschärft sich dieses Problem sogar noch, weil (getrennt übersetzte) Module nicht in gleichen Seiten abgespeichert sind, so daß bei einem großen Programm durchaus sehr viele, nur teilweise genutzte Seiten existieren können.

3. Das Ersetzungsproblem

Durch das Betriebssystem muß festgelegt werden, welche Daten aus dem Arbeitsspeicher in den Hintergrundspeicher ausgelagert werden müssen, um Platz für neu benötigte Daten zu schaffen. Es muß ein entsprechender Algorithmus implementiert sein, der auswählt, welche Seite bzw. welches Segment zuerst "geopfert", d.h. aus dem Hauptspeicher ausgelagert wird.

Bei Segmentierungsverfahren stellt sich oft das Problem der Ersetzung nicht, weil ein Programm gleichzeitig nur eine bestimmte, maximale Anzahl von Segmenten benutzen darf, z.B. ein Segment für den Programmcode und eines für die Programmdaten. Wird für ein Programm auf Anforderung ein neues Segment eingelagert, dann bedeutet dies stets die Auslagerung seines entsprechenden, zuvor benutzten Segmentes. Es ist jedoch auch möglich, daß das Betriebssystem genauso verfährt, wie es im folgenden für seitenorientierte Speicher beschrieben wird.

Bei Seitenwechselverfahren umfaßt die Arbeitsmenge eines Programms normalerweise recht viele Seiten, unter denen die zu opfernde Seite ausgewählt werden muß. Die dabei bekanntesten Ersetzungsstrategien (Verdrängungsstrategien), die dabei angewandt werden, sind:

FIFO (*first-in-first-out*) Es wird die Seite ersetzt, die sich am längsten im Arbeitsspeicher befindet.

LIFO (*last-in-first-out*) Es wird die zuletzt eingelagerte Seite ersetzt.

LRU (*least recently used*) Es wird die Seite ausgelagert, auf die die längste Zeit nicht mehr zugegriffen worden ist (vgl. auch Abschnitt 3.8).

LFU (*least frequently used*) Es wird die Seite ersetzt, auf die seit ihrer Einlagerung am seltesten zugegriffen wird.

Zusätzlich werden solche Seiten als "Opfer" bevorzugt, die während ihrer Einlagerungsdauer nicht verändert worden sind, was z.B. bei Seiten gewährleistet ist, die ausschließlich Programmcode enthalten. In diesem Fall sind die im Hauptspeicher befindlichen Seiten mit den auf dem Hintergrundspeicher befindlichen Seiten identisch und müssen nicht explizit zurückgeschrieben werden. Daten, die durch

den Prozeß geändert werden, müssen hingegen vor ihrer Ersetzung in den Hintergrundspeicher zurückgebracht werden.

4.2.2.2 Segmentierung versus Seitenaufteilung

Bei den neueren 32-bit-Mikroprozessoren scheint sich ein in Seiten aufgeteilter Speicher durchzusetzen. Zwar gibt eine Aufteilung des Programms in Seiten nicht – wie bei einer Segmentierung – die logische Struktur des Programms wieder, jedoch ist der Verwaltungsaufwand bei der Zuweisung von Hauptspeicherplatz deutlich geringer.

Ein wesentlicher Unterschied zwischen beiden Ansätzen besteht auch in der Häufigkeit der erforderlichen Datentransfers: Jedes Programm benutzt meist nur wenige der relativ großen Segmente, so daß auch nur selten – dann allerdings umfangreiche – Datentransfers erforderlich sind. Durch die relativ kleine Seitengröße besitzt ein Programm vergleichsweise sehr viel mehr Seiten, und es sind auch entsprechend mehr Seiteneinlagerungen erforderlich.

Allerdings kann beim Paging-Verfahren sehr viel besser dafür gesorgt werden, daß nur die aktuelle Arbeitsmenge eines Programms in den Hauptspeicher eingelagert wird. Dazu werden genau die Seiten eingelagert, auf die am häufigsten zugegriffen wird. Segmente hingegen werden durch den Compiler oder (Assembler-) Programmierer festgesetzt und sind oft so groß, daß sie den Lokalitätsbereich eines Programms wesentlich überschreiten. Besteht ein Programm z.B. nur aus einem Code- und einem Datensegment, so kann es nur vollständig in den Arbeitsspeicher eingelagert werden.

4.2.2.3 Virtuelle Speicherverwaltung in der µP-Technik

Es ist offensichtlich, daß zur Implementierung der oben aufgeführten Strategien eine Menge Zusatzinformationen benötigt werden. So erfordert z.B. das häufig benutzte LRU-Verfahren Kenntnisse über den letzten Zugriffszeitpunkt aller Seiten im Arbeitsspeicher. Eine solche Strategie kann nur durch Unterstützung seitens der Hardware effizient implementiert werden. Dies wird durch die MMUs der modernen Mikroprozessoren gewährleistet. Bei der Realisierung sind generell zwei verschiedene Wege begangen worden.

- Einerseits wird die MMU direkt auf dem Prozessorchip integriert. Dies ist z.B. bei den Intel-Prozessoren 80286 und 80386 sowie bei Zilogs Z8000 und Z800 der Fall.
- Andererseits wird die MMU in einem eigenen Baustein realisiert, so beim Motorola-Prozessor 68020 und National Semiconductors NS32332.

Der Transfer zwischen MMU und CPU ist natürlich bei einer auf dem Chip integrierten MMU schneller. Wird die MMU jedoch auf einen eigenen Chip realisiert, dann ist dafür auf dem Prozessorchip noch Platz für zusätzliche Erweiterungen. So konnte man beim Motorola 68020 z.B. einen Befehls-Cache auf dem Prozessorchip unterbringen (s. Abschnitt 3.8).

4.2.2.4 Zusammenspiel zwischen Cache und MMU

Im Abschnitt 3.8 hatten wir gezeigt, wie durch den Einsatz eines Cache-Speichers die mittlere Zugriffszeit zum Hauptspeicher erheblich reduziert werden kann. Ist im System eine MMU vorhanden, so stellt sich die Frage, wie zusätzlich ein Cache im System eingesetzt werden kann. Im Bild 4.2-3 sind zwei Möglichkeiten für die Zusammenarbeit von Cache und MMU dargestellt, die sich darin unterscheiden, ob die virtuelle oder die physikalische Adresse gespeichert wird. In beiden Fällen werden die Daten an der MMU vorbei zwischen der CPU, dem Cache und dem Speicher ausgetauscht.

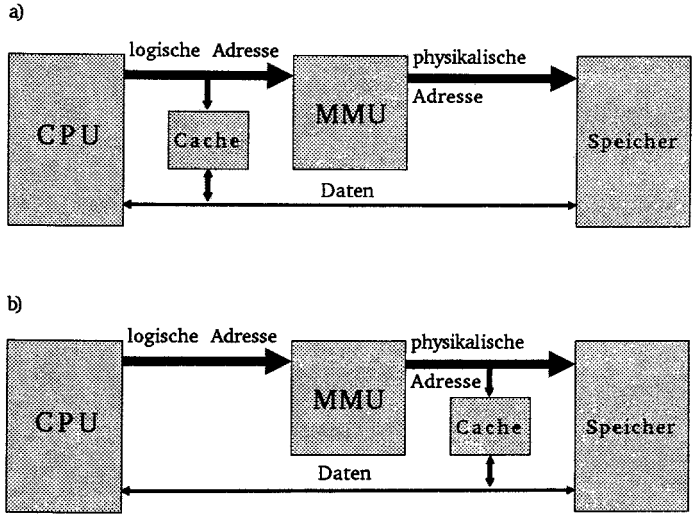


Bild 4.2-3. MMU und Cache;
a) virtueller Cache, b) physikalischer Cache

- Der **virtuelle Cache** (Bild 4.2-3a) wird zwischen CPU und MMU gelegt. In ihm werden die höherwertigen Bits der logischen Adressen als *Tags* abgelegt.
- Der **physikalische Cache** wird zwischen MMU und Speicher eingesetzt. In ihm werden die höherwertigen Bits der durch die MMU berechneten physikalischen Adressen gespeichert.

Der virtuelle Cache hat den Vorteil, daß bei den (meist mit hoher Wahrscheinlichkeit) auftretenden Treffern die MMU nicht benötigt wird und daher keine Verzögerung durch eine Adreßberechnung der MMU verursacht wird. Beim physikalischen Cache hingegen muß jede Adresse zunächst von der MMU bearbeitet werden. Als ein Nachteil des virtuelle Caches ist zu nennen, daß der logische Adreßraum in der Regel sehr viel größer ist als der physikalische und daher stets

mehr Bits einer Adresse als Tag gespeichert werden müssen als beim physikalischen Cache. Der zweite Nachteil tritt auf, wenn – wie bei den modernen Mikroprozessoren, z.B. dem Motorola 68040 und dem Intel 80486 – Cache und MMU auf dem Prozessorchip integriert sind. Hier besteht keine Möglichkeit, den virtuellen Cache außerhalb des Chips zu vergrößern und so eine höhere Trefferrate zu erzielen. Beim physikalischen Cache ist dies jedoch im Prinzip ohne weiteres möglich. Wohl aus diesem Grund verfügen die beiden eben erwähnten Prozessoren nur über einen physikalischen Cache.

Überblick

In den nächsten beiden Abschnitten werden die durch Mikroprozessoren unterstützten Speicherverwaltungskonzepte anhand dreier verschiedener Ansätze vorgestellt. Zunächst betrachten wir am Beispiel des Intel 80286 eine segmentorientierte Speicherverwaltung. Danach wird die Realisierung des Seitenwechselverfahrens genauer untersucht. Dabei wird zunächst auf den Intel 80386 eingegangen, bei dem die MMU auf dem Prozessorchip integriert ist und der sowohl eine Speichersegmentierung als auch eine Seitenverwaltung unterstützt. Anschließend wird gezeigt, wie eine in einem eigenen Baustein realisierte MMU, nämlich der Baustein NS32382 von National Semiconductor arbeitet.

4.3 Segmentorientierte Speicherverwaltung

In diesem Abschnitt soll gezeigt werden, wie die segmentorientierte Speicherverwaltung durch einen Mikroprozessor unterstützt werden kann. Dabei werden wir die Speicherverwaltung an einem typischen Vertreter vorstellen, dem 16-bit-Mikroprozessor Intel 80286, bei dem die MMU bereits auf dem Chip realisiert ist. Wenn nicht ausdrücklich auf andere Implementierungen verwiesen wird, beziehen sich alle Angaben dieses Abschnitts auf diesen Prozessortyp.

4.3.1 Segmente

Bei segmentorientierten Speicherverwaltungen ist, wie im letzten Abschnitt dargestellt, der virtuelle Adreßraum jedes Prozesses in ein oder mehrere physikalische Segmente unterteilt. Ein Segment ist ganz allgemein ein zusammenhängender Speicherbereich variabler Länge. Weil bei modernen Mikroprozessoren häufig Code und Daten strikt voneinander getrennt sind, wird zwischen verschiedenen Segmenttypen, z.B. Code- und Datensegmenten unterschieden. Die maximale Größe eines Segments liegt zwischen 64 kbyte beim Intel 80286 und 4 Gbyte bei modernen 32-bit-Prozessoren, wie z.B. dem 80386.

4.3.1.1 Berechnung physikalischer Adressen

Zunächst betrachten wir allgemein die Speicherabbildungs-Funktion für die segmentorientierte Speicherverwaltung. Ein Teil der virtuellen Adresse verweist auf das Segment, in dem sich das Speicherwort befindet, der restliche Teil bestimmt die genaue Position des Speicherwortes innerhalb des Segmentes.

Beim Intel 80286 wird ein Speicherwort durch eine virtuelle 32-bit-Adresse angesprochen, wie sie in Bild 4.3-1 schematisch dargestellt ist. Die höherwertigen 16 Bits bilden den sogenannten Segment-Selektor (vgl. Abschnitt 4.3.2.3), der die Anfangsadresse des Segmentes bestimmt, in dem sich das gesuchte Speicherwort befindet. Die niederwertigen 16 Bits legen als Offset die genaue Position des Wortes innerhalb des selektierten Segments fest.

Alle vom Mikroprozessor benutzten Speicheradressen müssen von dieser Form sein, d.h. aus einem Selektor und einem Offset bestehen. Wegen der Lokalitätseigenschaft (vgl. Abschnitt 4.2) wird normalerweise nicht bei jedem Zugriff zum Arbeitsspeicher ein neues Segment benutzt. Daher werden für längere Zeit dieselben Segment-Selektoren benötigt, und es ist aus Geschwindigkeitsgründen sinnvoll, diese in speziellen Segment-Registern abzuspeichern. Ein Speicherwort wird dann nicht durch die Angabe der vollständigen virtuellen Adresse im Befehl adressiert, sondern es genügt, im Befehl das Segmentregister und den Offset anzugeben. Eine übliche Assemblerschreibweise dafür ist:

<Mnemocode> <Segmentregister>:<Offset> .

Die Adressierung läßt sich dadurch noch weiter erleichtern, daß jedes der Segmentregister für bestimmte Segmenttypen (Codesegment, Datensegment, etc.) zuständig ist. Dadurch kann bei den meisten Speicherzugriffen auf eine explizite Angabe des Segmentregisters verzichtet werden, denn der Prozessor erkennt anhand des Befehls, ob z.B. ein Zugriff auf einen Befehlscode oder einen Operanden stattfindet. Dies verkürzt die Assemblerschreibweise zu:

<Mnemocode> <Offset> .

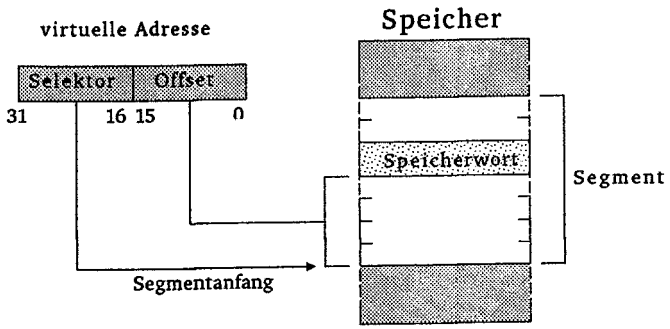


Bild 4.3-1. Darstellung der Adressierung bei einem segmentierten Speicher

4.3.1.2 Fallstudie: Speicherverwaltung beim Intel 80286

Der Intel 80286 stellt dem Benutzer die in Bild 4.3-2 dargestellten, vier verschiedenen Segmentregister zur Verfügung (vgl. auch Abschnitt 1.7). Immer dann, wenn ein neuer Selektor in eines der Segmentregister geschrieben wird, werden automatisch vom Prozessor wichtige Informationen über das ausgewählte Segment, die in den sogenannten Segment-Deskriptoren abgelegt sind, in den zugeordneten Teil des Segment-Deskriptor-Caches geschrieben (*hidden descriptor cache*). (Auf Form und Inhalt der Segment-Deskriptoren gehen wir im Abschnitt 4.3.2 ausführlich ein.) Durch diesen Cache kann der Prozessor ein Segment bearbeiten, ohne für jeden Befehl zeitaufwendig auf den Segment-Deskriptor im Arbeitsspeicher zugreifen zu müssen.

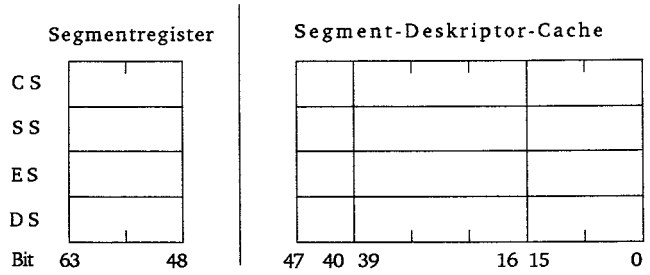


Bild 4.3-2. Segmentregister und Segment-Deskriptor-Cache beim Intel 80286

CS-Register

Jede Instruktion eines Prozesses ist in einem **Codesegment** (*code segment*) abgespeichert. Das CS-Register enthält den Selektor für das Segment, in dem der aktuell benutzte Programm-Code abgelegt ist. Beim Laden eines Befehls (*instruction fetch, opcode fetch*) wird automatisch der Selektor aus diesem Register benutzt. Der Offset des gewünschten Befehls zum Segmentanfang wird durch den Befehlszähler IP (*instruction pointer*) spezifiziert. Die Konkatenation der Registerinhalte CS und IP, in Assemblerschreibweise "CS:IP", gibt die vollständige 32-bit-Adresse für den neu zu ladenden Befehlscode. Eine Änderung des Inhalts des CS-Registers und damit ein Wechsel zu einem neuen Codesegment kann nur indirekt durch *Interrupts*, *Traps*, *Exceptions* oder durch Sprünge zu virtuellen Adressen in anderen Segmenten (*jump far*) erfolgen. Das Segmentregister CS kann also nicht direkt, beispielsweise durch einen Transfer-Befehl (MOVE[1)]), neu gesetzt werden.

Im Bild 4.3-3 ist die Adressierung der verschiedenen Segmente, die wir im folgenden beschreiben werden, skizziert.

1) Intel-Bezeichnung: MOV

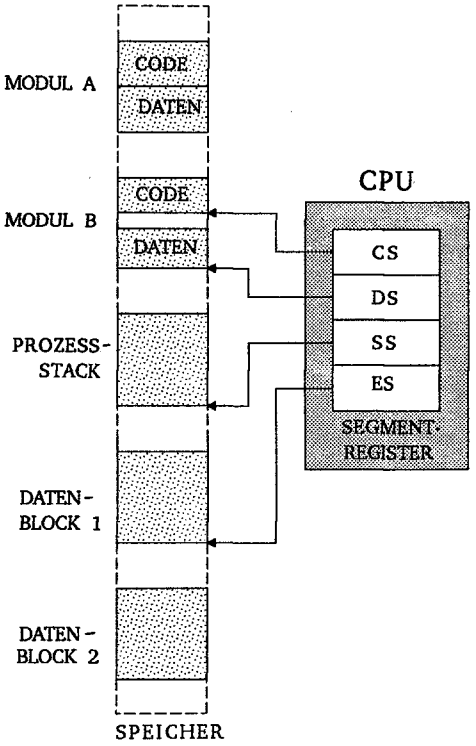


Bild 4.3-3. Zugriff auf die Arbeitsmenge eines Prozesses mit Hilfe der Segmentregister

SS-Register

Fast jeder Prozeß benötigt einen Stack, z.B. für Unterprogramm-Aufrufe. Auch ein Stack beansprucht Speicherplatz, für den ebenfalls ein bestimmtes Segment, das **Stacksegment** (*stack segment*), im Speicher vorgesehen ist. Dieses Segment wird durch den Selektor im SS-Register spezifiziert. Alle Stack-Operationen beziehen sich automatisch auf dieses Segment. Der Adressen-Offset zum Segmentanfang wird durch den Stackpointer SP festgelegt. Das SS-Register läßt sich (im Gegensatz zu CS) auch explizit neu laden, z.B. durch den Befehl: MOVE SS,AX, wobei AX den (16-bit-)Akkumulator des 80286 bezeichnet (s. Abschnitt 1.7).

DS-Register und ES-Register

Jedes Programm benutzt selbstverständlich auch noch Daten. Die Register DS und ES ermöglichen es, zwei verschiedene **Datensegmente** (*data segment, extra data segment*) zu spezifizieren, auf die der Benutzer unter Angabe der Segmentregister direkt zugreifen kann. Wenn kein Datensegment-Register explizit angege-

ben ist, werden alle Datenzugriffe standardmäßig auf das Register DS bezogen. Lediglich Ziele von Zeichenketten-Operationen (*strings*) beziehen sich stets auf das durch ES spezifizierte Segment.

Beispiel
MOVSW kopiert das Speicherwort an der Stelle DS:[SI] nach ES:[DI], wobei [SI], [DI] die indirekte Registeradressierung mit den Indexregistern SI, DI kennzeichnen (vgl. Abschnitt 1.15).

Um zusätzlich noch weitere Datensegmente benutzen zu können, lassen sich das DS- und das ES-Register explizit durch MOVE-Befehle laden. Bei anderen Mikroprozessoren, z.B. dem Intel 80386, gibt es z.T. noch zusätzliche Datensegment-Register.

Jedes Segment, dessen Anfangsadresse in einem der Segmentregister abgelegt ist, befindet sich physikalisch im Hauptspeicher, so daß darauf unverzüglich zugegriffen werden kann. Somit bilden die durch die Segmentregister spezifizierten Segmente die Arbeitsmenge (*working set*) des Prozesses (s. Abschnitt 4.2).

4.3.1.3 Adressierungs-Modi

In Bild 4.3-1 wurde nur schematisch dargestellt, wie mit einem Segment-Selektor und einem Offset ein Speicherwort ausgewählt werden kann. Wir wollen nun genauer darstellen, wie mit Hilfe dieser beiden Größen physikalische Adressen generiert werden. Um Aufwärts-Kompatibilität mit den älteren Mikroprozessoren innerhalb der Produktfamilie zu gewährleisten, können viele Prozessoren in verschiedenen Adressierungs-Modi arbeiten.

So kann der Intel 80286 in zwei unterschiedlichen Adressierungs-Modi betrieben werden, dem *Real Address Mode* und dem *Protected Virtual Address Mode*. Damit der Prozessor unterscheiden kann, in welchem Modus er sich gerade befindet, ist im letztgenannten Modus in seinem Statusregister MSW (*machine status word*) ein entsprechendes Bit gesetzt (*protection enable* – PE, vgl. Abschnitt 1.7).

Real (Address) Mode

Da in diesem Modus die Adressierungsfähigkeiten des Intel 8086 maßgebend sein sollen, sind nur 20 bit lange physikalische Adressen zugelassen, d.h. die maximal adressierbare Hauptspeichergröße beträgt 1 Mbyte. Wie aus dem 16-bit-Selektor und dem 16-bit-Offset eine 20 bit lange physikalische Adresse gebildet wird, zeigt Bild 4.3-4.

Der Selektor gibt die 16 höherwertigen Bits der 20 bit langen Segment-Basisadresse (Anfangsadresse des Segments) an. Die verbleibenden 4 Bits werden als '0000' angenommen. Durch den 16 bit langen Offset können Segmente einer maximalen Länge von 64 kbyte angesprochen werden. Die physikalische Adresse errechnet sich durch die Addition von Segment-Basisadresse und Offset.

In diesem Modus können alle auf dem 8086 entwickelten Programme verarbeitet werden. Allerdings sind die erweiterten Möglichkeiten und Fähigkeiten des 80286, z.B. zur Speicherverwaltung, nicht nutzbar.

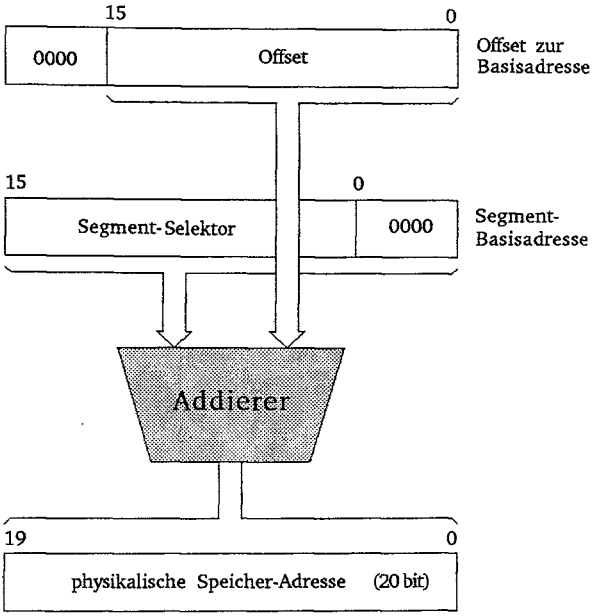


Bild 4.3-4. Adreßberechnung des 80286 im Real Address Mode

Protected (Virtual Address) Mode

Im folgenden werden wir uns ausschließlich mit diesem Adressierungs-Modus beschäftigen, in dem alle erweiterten Möglichkeiten des 80286 voll ausgeschöpft werden können, d.h. in diesem Modus stehen die erweiterte Speicherverwaltung, die Multitasking-Fähigkeit und die implementierten Schutzmechanismen zur Verfügung. So kann z.B. von der Speicherverwaltung ein 1 Gbyte großer virtueller Adreßraum in einen maximal 16 Mbyte großen physikalischen Adreßraum abgebildet werden. Die Begrenzung des physikalischen Adreßraums ergibt sich hier aus den 24 Adreßleitungen, die beim 80286 zur Verfügung stehen. In Bild 4.3-5 ist das Adressierungsverfahren dargestellt.

Im *Protected Mode* spezifiziert der Selektor-Teil der virtuellen Adresse nicht direkt die Basisadresse des Segments, sondern verweist auf den Eintrag einer im Speicher vorhandenen Tabelle. Dieser Eintrag heißt Segment-Deskriptor und enthält u.a. die 24 bit lange Basisadresse des Segments. Zu dieser Basisadresse wird, wie im *Real Mode*, der Offset addiert, um die physikalische Adresse zu erhalten. Die Wahl der (richtigen) Tabelle, der Segment-Deskriptor-Tabelle, erfolgt automatisch durch die CPU; es ist dazu keine zusätzliche Software erforderlich.

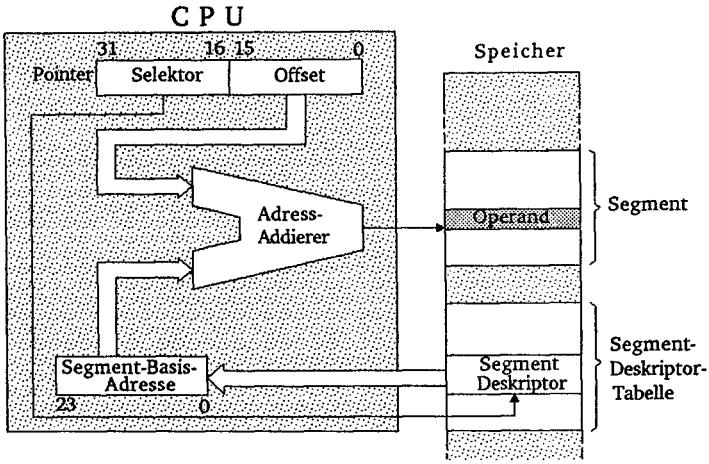


Bild 4.3-5. Adreßberechnung im Protected Virtual Address Mode beim Intel 80286

Wir wollen uns jetzt die Einträge in diesen Tabellen, also die Segment-Deskriptoren, genauer ansehen. Die Deskriptoren bilden die Basis der virtuellen Speicherverwaltung und der Schutzmechanismen.

4.3.2 Segment-Deskriptoren und Deskriptor-Tabellen

4.3.2.1 Segment-Deskriptoren

Jedes Segment kann durch drei Attribute beschrieben werden:

- die Segment-Basisadresse (*base address*),
- die Segmentgröße in byte (*limit*),
- die Zugriffsrechte (*access rights*) zur Realisierung von Schutzmechanismen (*protections*).

Diese Attribute sind in den **Segment-Deskriptoren** abgelegt, die den Segmentzugriff steuern. Im folgenden wollen wir als Beispiel den 80286-Prozessor genauer betrachten, bei dem die Deskriptoren 8 byte lang sind und das im Bild 4.3-6 dargestellte Format haben. Die Bytes 6 und 7 sind für den Intel 80386 reserviert. Auf sie gehen wir im Abschnitt 4.4.1 näher ein. Sie müssen beim 80286 auf '0' gesetzt werden. Die unteren beiden Bytes geben die Segmentgröße (*limit*) an. Es gilt:

kleinste Segment-Adresse = Basisadresse
größte Segment-Adresse = Basisadresse + Segmentgröße.

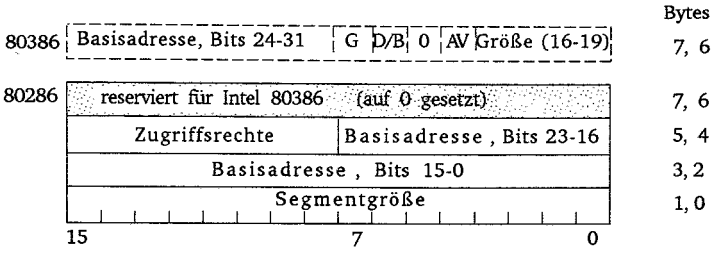


Bild 4.3-6. Format des Segment-Deskriptors

Die Segmentgröße ist also der größtmögliche Offset. Die Gesamtgröße des Segments ist gegeben durch: Segmentgröße + 1. Weil 16 Bits zur Verfügung stehen, ist eine maximale Segmentgröße von 2^{16} byte (64 kbyte) möglich. Die Basisadresse besteht aus 3 Bytes und erlaubt so die vollständige Adressierung des maximalen, physikalischen Adreßraums von 16 Mbyte (2^{24} byte). Im fünften Byte (*access byte*) werden die Zugriffsrechte spezifiziert, wobei die einzelnen Bits die im folgenden beschriebene Bedeutung haben (vgl. Bild 4.3-7).

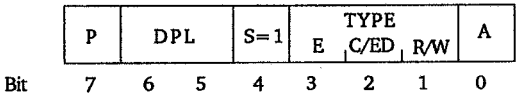


Bild 4.3-7. Zugriffsrechte im Segment-Deskriptor

P (*present bit*) Dieses Bit zeigt an, ob sich das durch den Deskriptor beschriebene Segment im Hauptspeicher befindet. Ist P=0, ist es nicht speicherresident und muß vom Betriebssystem in den Hauptspeicher eingelagert werden, bevor auf Daten aus diesem Segment zugegriffen werden kann (vgl. Abschnitt 4.2).

DPL (*Descriptor Privilege Level*) Eine der wichtigsten Erweiterungen zur Implementierung von Schutzmaßnahmen ist die Einführung spezieller Prioritäten, die bei Intel Privileg-Ebenen (*privilege levels* – PL) genannt werden. Sie ermöglichen, daß Segmente vor unzulässigem Zugriff direkt durch die Hardware geschützt werden. Bei den Intel-Prozessoren gibt es die vier Privileg-Ebenen 0 - 3. Sowohl Segmente als auch Prozesse gehören stets eindeutig zu einer bestimmten Privileg-Ebene. Es gibt genaue Zugriffsregeln, die festlegen, welche Segmente von einem bestimmten Prozeß benutzt werden dürfen. Genaueres zu diesem Problemkreis wird im Abschnitt 4.5 erklärt.

TYPE Diese drei Bits unterscheiden die verschiedenen Segmenttypen. Dabei bedeuten im einzelnen:

E (*executable bit*, Bit 3) Mit diesem Bit läßt sich feststellen, ob es sich um ein Datensegment (E=0) oder ein Codesegment (E=1) handelt.

C/ED (*conforming bit* bzw. *expand down bit*, Bit 2)

- Beschreibt der Deskriptor ein Codesegment und ist das Bit 2 gesetzt, d.h. C=1, dann ist es ein sogenanntes *Conforming Code Segment*. Segmente diesen Typs können auch von Prozessen mit einer niedrigeren Privileg-Ebene benutzt werden. Gilt C=0, so handelt es sich um ein *Nonconforming Code Segment*. Genaueres dazu finden Sie ebenfalls im Abschnitt 4.5.
- Beschreibt der Deskriptor ein Datensegment, dann bedeutet das gesetzte Bit 2, d.h. ED=1, daß es sich um ein *Expand Down Segment* handelt. Solche Segmente werden zur Realisierung von Stacks benutzt. (Der Begriff *expand down* bedeutet gerade, daß dieses Segment sich nach unten, also zu kleineren Adressen hin, ausdehnt.) Bei ihnen darf der Offset größer als die maximale Segmentgröße werden. Die Adressierung verläuft anders als bei normalen Segmenten, und zwar gilt:

  kleinste Segmentadresse = Basisadresse + Segmentgröße + 1,
  größte Segmentadresse = Basisadresse + 64 kbyte
  (Segmentgröße: $FFFF).

  Im weiteren wollen wir nicht weiter auf Expand-Down-Segmente eingehen.

R/W (*read/write bit*, Bit 1)

- Bei einem Codesegment (E=1) bedeutet ein gesetztes Bit 1, d.h. R=1, daß in diesem Segment enthaltene Daten gelesen werden dürfen, andernfalls gilt R=0.
- Bei einem Datensegment legt das Bit 1 fest, ob in das Segment neue Daten eingeschrieben werden dürfen (W=1) oder nicht (W=0).

A (*accessed bit*, Bit 0) Dieses Bit wird nach jedem Zugriff auf das Segment gesetzt. Das Betriebssystem kann das *Accessed Bit* in bestimmten Zeiträumen wieder zurücksetzen und so feststellen, wie häufig auf einzelne Segmente zugegriffen wurde. Mit Hilfe dieses Bits wird für das Betriebssystem die Implementierung von Ersetzungsstrategien wie z.B. LRU erleichtert (vgl. Abschnitt 4.2).

S Dieses Bit (Bit 4) ist hier auf '1' gesetzt. Dadurch wird gekennzeichnet, daß es sich um einen Segment-Deskriptor handelt. Bei anderen Deskriptor-Arten ist das S-Bit auf '0' gesetzt. Diese sogenannten Kontroll-Deskriptoren werden Sie in den folgenden Abschnitten kennenlernen.

Die oben vorgestellten Deskriptoren werden in Tabellen abgelegt und verwaltet. Mit diesen Tabellen werden wir uns nun beschäftigen.

4.3.2.2 Deskriptor-Tabellen

Für jeden Prozeß gibt es eine lokale ("private") Deskriptor-Tabelle. Des weiteren gibt es für die Segmente, die von allen Prozessen gemeinsam benutzt werden dürfen, eine globale Deskriptor-Tabelle. Da jeder Deskriptor 8 byte lang ist und die maximale Segmentgröße (beim 80286) 64 kbyte beträgt, kann jede Tabelle maximal 8k (=8192) Deskriptoren enthalten.

Globale Deskriptor-Tabelle (GDT)

Die GDT (*global descriptor table*) enthält – wie eben gesagt – die Deskriptoren derjenigen Segmente, die von allen Prozessen gemeinsam benutzt werden dürfen. Dies können z.B. Segmente mit dem Betriebssystemcode, Compilern, Editoren oder ähnliche, für viele Prozesse wichtige Dienste sein. Auch die globale Deskriptor-Tabelle ist letztendlich ein Segment, das sich an einer beliebigen Stelle im Hauptspeicher befinden darf. Weil aber in jedem System nur eine einzige GDT vorhanden ist, ist zu ihrer Spezifikation kein spezieller Deskriptor erforderlich. Der Zugriff auf diese Tabelle (dieses Segment) erfolgt stattdessen über ein spezielles Register im Prozessor, dem sogenannten globalen Deskriptor-Tabellen-Register (*Global Descriptor Table Register* – GDTR). Es ist ein 40-bit-Register mit der im Bild 4.3-8 angegebenen Form.

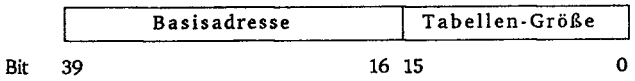


Bild 4.3-8. GDT-Register

Die höchstwertigen drei Bytes bestimmen die Basisadresse, die letzten zwei Bytes die Größe der GDT. Das *Access Byte* fehlt, weil alle Prozesse das Zugriffsrecht auf die GDT besitzen.

Lokale Deskriptor-Tabelle (LDT)

Die LDT (*local descriptor table*)enthält die Segment-Deskriptoren der "privaten" Code- und Daten-Segmente eines Prozesses, auf die andere Prozesse keinen Zugriff erhalten sollen. Jeder Prozeß besitzt also seine eigene LDT. Obwohl somit ein anderer Prozeß normalerweise nicht auf Segmente aus einer "fremden" LDT zugreifen kann, gibt es spezielle Mechanismen, die den kontrollierten Zugriff auf fremde Segmente erlauben (s. Abschnitt 4.7).

Verwaltung der Deskriptor-Tabellen

Sobald ein neuer Prozeß generiert wird, erzeugt das Betriebssystem für ihn eine LDT. Dort trägt es alle Deskriptoren ein, die die Segmente des Prozesses beschreiben. In diesen Deskriptoren werden die *Present Bits* P zunächst zurückgesetzt und zeigen dadurch an, daß sich die Segmente noch nicht im Hauptspeicher befinden. Zusätzlich erzeugt das Betriebssystem in der GDT einen Deskriptor, der auf die LDT verweist.

Wenn der Prozeß im Zustand "aktiv" ist, werden vom Betriebssystem die vom Prozeß aktuell benötigten Segmente vom Hintergrundspeicher in den Hauptspeicher geladen. Anhand der momentanen Hauptspeicher-Belegung bestimmt es für jedes Segment eine geeignete Speicherlücke und legt dadurch die Basisadresse fest. Diese wird im Segment-Deskriptor eingetragen und das Segment durch das P-Bit als speicherresident gekennzeichnet.

Die vom System verwalteten lokalen Deskriptor-Tabellen sind spezielle Segmente, die ausschließlich Deskriptoren enthalten. Wie jedes Segment können auch die LDTs verschiedene Größen aufweisen und sind, wie z.B. Code- und Daten-Segmente, durch

- die Basisadresse (*base address*),
- die Größe (*limit*) und
- die Zugriffsrechte (*access rights*)

gekennzeichnet. Im Gegensatz zur GDT-Tabelle, die es ja nur einmal gibt und auf die alle Prozesse zugreifen dürfen, besitzt eine LDT also auch spezielle Zugriffsrechte. Die oben genannten Kenngrößen einer LDT sind in einem speziellen LDT-Deskriptor abgespeichert, der das in Bild 4.3-9 dargestellte Format besitzt. Damit der Prozessor die Basisadressen der benötigten LDTs finden kann, müssen sämtliche LDT-Deskriptoren in der GDT abgespeichert sein.

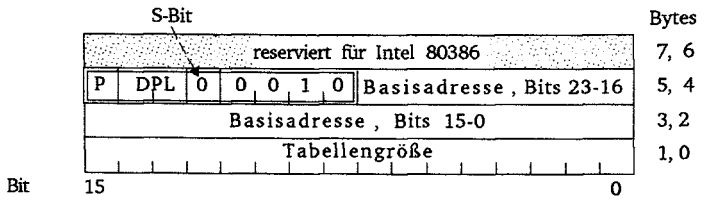


Bild 4.3-9. LDT-Deskriptor

Die im doppelt umrandeten (*Access*-)Byte eingetragenen Zugriffsrechte haben dieselbe Bedeutung wie bei den oben beschriebenen Segment-Deskriptoren. Die letzten, auf "00010" gesetzten Bits dieses Bytes identifizieren den Deskriptor als einen LDT-Deskriptor. Wie bereits erwähnt, zeigt dabei das auf '0' gesetzte S-Bit, daß es sich um einen Kontroll-Deskriptor handelt.

Für den Zugriff auf die LDT des augenblicklich aktiven Prozesses gibt es im Prozessor ein Register, das sogenannte LDT-Register (*Local Descriptor Table Register* – LDTR), das das im Bild 4.3-10 dargestellte Format besitzt.

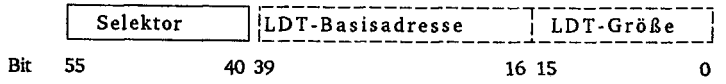


Bild 4.3-10. Das LDT-Register

Durch einen Befehl kann nur auf die 16 höchstwertigen Bits des Registers direkt zugegriffen werden, die den Selektor des aktuellen LDT-Deskriptors in der GDT enthalten. Wird nach einem Prozeßwechsel ein neuer Selektor dort eingetragen, so lädt der Prozessor automatisch aus der GDT die Basisadresse und die Größe der neuen LDT in den niederwertigen, "unsichtbaren" Teil des LDTR-Registers (*invisible, hidden descriptor cache*). Dadurch wird für alle folgenden Zugriffe auf die LDT das zeitaufwendige Lesen ihres Deskriptors aus der GDT vermieden.

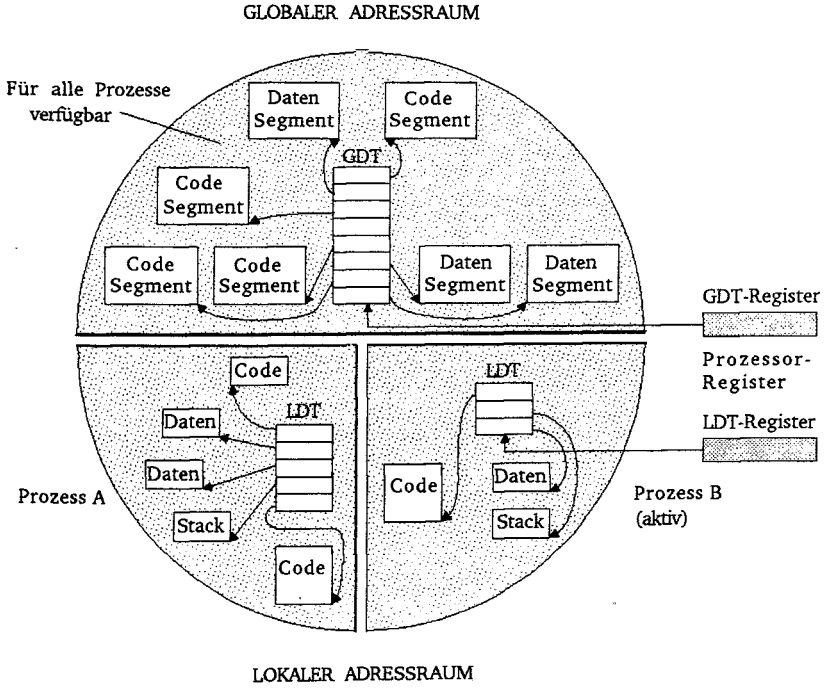


Bild 4.3-11. Aufteilung der Datenbereiche zweier Prozesse A, B durch GDT und LDT

In Bild 4.3-11 (s.o.) wird dargestellt, wie die lokalen Datenbereiche zweier Prozesse A und B voneinander getrennt werden. Auf den globalen Adreßraum können beide gemeinsam zugreifen. Das LDT-Register zeigt im dargestellten Beispiel auf die LDT des Prozesses B, d.h. B ist der momentan vom Prozessor bearbeitete Prozeß.

Selbsttestaufgabe S4.3-1:
a) Wie groß ist beim Intel 80286 die maximal mögliche Anzahl von Segmenten für einen einzelnen Prozeß ?
b) Wie groß ist maximal der virtuelle Adreßraum eines Prozesses ?
c) Wie groß ist maximal der physikalische Adreßraum ?

4.3.2.3 Selektoren

Nachdem wir die im System zur Speicherverwaltung vorhandenen Tabellen beschrieben haben, können wir uns die Berechnung physikalischer Adressen genauer ansehen. In Bild 4.3-5 wurde dargestellt, wie mit Hilfe des Selektor-Teils einer virtuellen Adresse der Zugriff auf eine Segment-Deskriptor-Tabelle erfolgte. Inzwischen wissen wir, daß in dieser Tabelle die Deskriptoren mit allen für die Speicherverwaltung erforderlichen Informationen abgelegt sind. Der Aufbau des Selektor-Teils jeder virtuellen Adresse ist im Bild 4.3-12 skizziert.

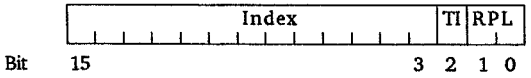


Bild 4.3-12. Selektor einer virtuellen Adresse

Das Indexfeld und der Tabellenindikator TI spezifizieren den Eintrag in einer Deskriptor-Tabelle, in dem für das gesuchte Segment der Deskriptor abgelegt ist. TI bestimmt, ob die virtuelle Adresse zum globalen oder lokalen Adreßraum gehört. Ist das TI-Bit gesetzt, d.h. TI=1, dann wird die LDT benutzt; gilt TI=0, dann ist der gesuchte Deskriptor in der GDT zu finden. Der Index gibt für das ausgewählte Segment die Nummer des Eintrags in der Deskriptor-Tabelle an. Dabei wird mit dem Indexwert 0 begonnen. (Eine Index-Länge von 13 bit entspricht 8192 Möglichkeiten.)

Auch ein Selektor besitzt eine Privileg-Ebene, die sogenannte verlangte Privileg-Ebene (*Requested Privilege Level* – RPL). Ob überhaupt ein Zugriff auf ein

bestimmtes Segment erfolgen darf, muß der Prozessor zunächst noch prüfen. RPL muß dazu in Abhängigkeit von der Art des Segmentes (Code oder Daten) in einer vorgegebenen Relation zur Privileg-Ebene des gerade ausgeführten Prozesses stehen. Genauer wird dies im Abschnitt 4.5 beschrieben.

Selbsttestaufgabe S4.3-2:

a) Geben Sie einen Deskriptor an, der ein *Nonconforming Code Segment* beschreibt, das folgende Eigenschaften hat:

   - Privileg-Ebene: 3,
   - Größe: 8 kbyte,
   - Anfangsadresse: $10300,
   - Lage: nicht im Arbeitsspeicher,
   - Zugriffsrechte: Daten dürfen gelesen werden.

b) Bestimmen Sie die virtuelle (logische) Adresse einer Variablen, die sich mit einem Offset von $0FA3 in einem Code-Segment der Privileg-Ebene PL=3 befindet! Der zugehörige Deskriptor liege in GDT(5), d.h. im Eintrag mit der Nummer 5 der GDT.

   Wie lautet (in Hexadezimal-Schreibweise) die zugehörige physikalische Adresse, wenn in GDT(5) der in Teil a) bestimmte Deskriptor steht ?

4.4 Seitenorientierte Speicher-Verwaltung

In diesem Abschnitt wird anhand zweier Beispiele demonstriert, wie die Speicherverwaltungs-Einheiten (MMUs) für einen seitenorientierten Speicher organisiert sind. Zunächst zeigen wir am Beispiel des 32-bit-Prozessors Intel 80386, wie eine auf dem Mikroprozessor-Chip realisierte Seitenverwaltung arbeitet. Dieser Prozessortyp unterstützt sowohl die Segmentierung wie auch die Seitenspeicher-Verwaltung. Anschließend wird die in einem diskreten Baustein realisierte MMU NS32382 von National Semiconductor vorgestellt, die ausschließlich die Verwaltung von Seiten unterstützt.

4.4.1 Seitenverwaltungskonzept des Intel 80386

Der 32-bit-Prozessor Intel 80386 bietet annähernd die gleichen Konzepte zur Speichersegmentierung wie der im vorigen Abschnitt vorgestellte 80286, aber zusätzlich noch die Möglichkeit einer seitenorientierten Speicherverwaltung. Die Speicheraufteilung in Seiten ist dabei optional, d.h. der Prozessor kann auch aus-

schließlich mit Segmenten arbeiten, so daß die Kompatibilität mit dem Prozessor 80286 gewährleistet ist. Die erforderlichen Erweiterungen zur Verwaltung von Speicherseiten werden im folgenden näher beschrieben.

Wie bereits erwähnt, ist bei einer Seitenverwaltung der Speicher in Bereiche gleicher Länge, die sogenannten Seiten (*pages*), unterteilt. Dies hat Konsequenzen für die Adressierung:

Während Segmente an jeder beliebigen Stelle im Hauptspeicher beginnen können, ist das bei Speicherseiten nicht möglich. Der gesamte Speicher wird bei der Adresse 0 beginnend in ein Raster aus Seiten der Größe 4 kbyte (2^{12} byte) unterteilt. Deshalb sind bei den Basisadressen aller Seiten die niederstwertigen 12 Bits auf 0 gesetzt.

Diese Einschränkung vereinfacht die Verwaltung des Speichers sowie die Adreßberechnung erheblich. Im Gegensatz zu den Segmenten finden sich jedoch – wie bereits erwähnt – für einzelne Seiten keine Entsprechungen in der logischen Struktur des Programms. Allerdings kann aber bei Seiten fester Länge die Lokalitätseigenschaft von Programmen effizienter ausgenutzt werden.

4.4.1.1 Berechnung physikalischer Adressen aus virtuellen Adressen

Die Seitenverwaltung des 80386 ist dem segmentorientierten Organisationskonzept des 80286 aufgesetzt. Deshalb wird zwischen drei verschiedenen Adreß-Arten unterschieden: virtuelle (logische), lineare und physikalische Adressen.

Virtuelle Adressen bestehen aus einem 16-bit-Selektor und einem 32-bit-Offset. Wie beim Segmentkonzept im vorigen Abschnitt vorgestellt wurde, werden die virtuellen Adressen mit Hilfe von Segment-Deskriptoren in 32 bit lange **lineare Adressen** umgesetzt. Beim 80386 enthalten die beiden höchstwertigen Bytes der Deskriptoren die höherwertigen 8 Bits der bei diesem Prozessor (von 24 bit beim 80286) auf 32 bit verlängerten Basisadresse des Segments, 4 zusätzliche Bits zur Erhöhung der maximalen Segmentgröße auf 2^{20} sowie einige Steuerbits. Von diesen wollen wir nur das Bit G (*granularity*) erwähnen, das die Wahl zwischen den Einheiten 'byte' und '4 kbyte' für die Segmentgröße erlaubt. Daraus ergibt sich eine maximale Segmentgröße von 1 Mbyte (2^{20} byte), falls die Einheit 'byte' gewählt ist, und 4 Gbyte ($2^{20} \cdot 4$ kbyte = 2^{32} byte), falls die Einheit '4 kbyte' ist.

Die Deskriptoren sind in den Deskriptor-Tabellen abgelegt. Werden keine Speicherseiten verwendet, dann ist beim 80386 diese lineare Adresse auch gleichzeitig die physikalische Adresse. Ist jedoch der Speicher zusätzlich in Seiten aufgeteilt, was durch das Bit PG (*paging enabled*, s. Abschnitt 1.7.3) im Maschinenstatuswort MSW festgelegt wird, dann werden die linearen Adressen durch die auf dem Prozessor integrierte Paging-Einheit in **physikalische Adressen** umgesetzt (s. Bild 4.4-1). Dazu dienen die Seitentabellen, die wir im folgenden beschreiben werden.

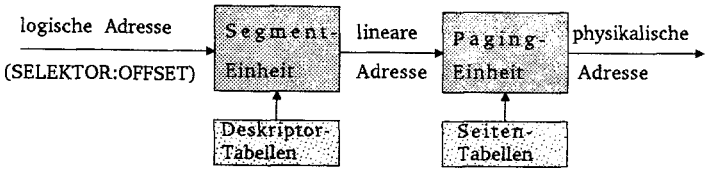


Bild 4.4-1. Berechnung physikalischer Adressen beim 80386

Das Bild 4.4-2 zeigt noch einmal genauer, wie virtuelle Adressen zunächst mit Hilfe der Segmenttabellen und dann mit einer zweistufigen Seitentabellen-Verwaltung in physikalische Adressen umgewandelt werden. Die prinzipielle Gewinnung der linearen Adresse wurde bereits im letzten Abschnitt beschrieben. Im Unterschied zum 80286 ist sie beim 80386 nicht nur 24 bit, sondern 32 bit lang (s.o.).

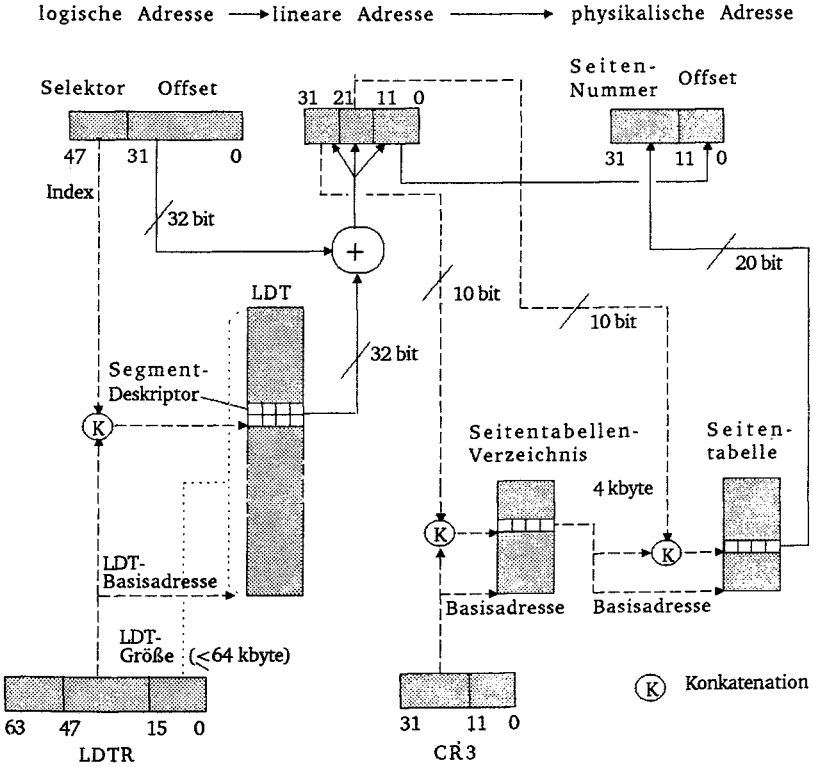


Bild 4.4-2. Berechnung physikalischer Adressen mit Hilfe von Segment- und Seitentabellen beim 80386

Die für uns neue Umsetzung der linearen Adresse durch die Paging-Einheit betrachten wir im folgenden Unterabschnitt genauer.

4.4.1.2 Berechnung physikalischer Adressen aus linearen Adressen

Zur Bestimmung physikalischer Speicheradressen wird, wie auch in den meisten Großrechnern, ein zweistufiges, hierarchisches Tabellen-Verfahren angewendet. Zur Adreßberechnung werden dabei das Seitentabellen-Verzeichnis (*page directory*), die Seitentabellen (*page tables*) und die Seiten (*pages*) selbst benötigt. Beim 80386 ist jede Tabelle und jede Seite genau 4 kbyte groß. Jeder (im folgenden noch genauer beschriebene) Tabellen-Eintrag umfaßt 4 byte, so daß in jeder Tabelle genau 1024 (teilweise auch ungenutzte) Einträge enthalten sind, die von 0 bis 1023 durchnumeriert sind. Bild 4.4-3 veranschaulicht noch einmal die Umsetzung linearer Adressen in physikalische Adressen.

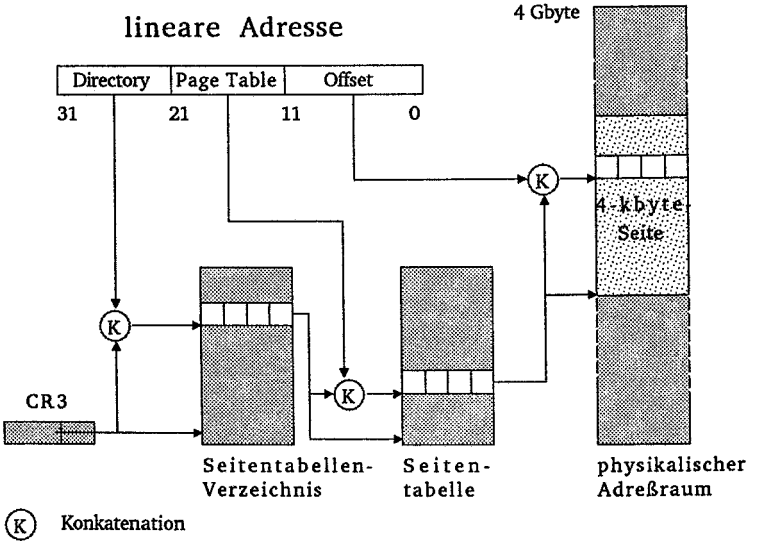


Bild 4.4-3. Umsetzung linearer in physikalische Adressen mit Hilfe von Seitentabellen

Bevor wir ausführlicher auf den Aufbau der Tabellen, ihre Funktion sowie den Aufbau der einzelnen Tabellen-Einträge eingehen, wollen wir ganz kurz die Ermittlung der physikalischen Adresse beschreiben.

Wie aus der Abbildung ersichtlich, wird jede lineare Adresse logisch in drei verschiedene Teile zerlegt. Auf ein bestimmtes Seitentabellen-Verzeichnis wird über ein spezielles Systemregister CR3 zugegriffen, in dem seine Basisadresse abgelegt ist. Die höchstwertigen 10 Bits der linearen Adresse selektieren einen Eintrag in diesem Verzeichnis. Der Eintrag seinerseits enthält die Adresse einer Sei-

tentabelle. Die nächsten 10 Bits der linearen Adresse selektieren einen der 1024 Einträge aus der Seitentabelle. In diesem Eintrag steht die Basisadresse einer bestimmten Seite. Die niederstwertigen 12 Bits der linearen Adresse werden als Offset zur Seitenadresse addiert, um so die endgültige, physikalische Adresse zu erhalten.

Das Seitentabellen-Verzeichnis

Jeder Prozeß hat sein eigenes **Seitentabellen-Verzeichnis** (*page directory*). Die Basisadresse dieses Verzeichnisses ist im Systemregister CR3 der CPU abgespeichert. Das Register CR3 ist eines von vier im 80386 vorhandenen 32-bit-Registern zur Seitenverwaltung. In CR3 sind nur die 20 höchstwertigen Bits signifikant, d.h. die niederstwertigen 12 Bits sind stets auf 0 gesetzt. (Dies ist, wie oben bereits bemerkt, die Form jeder Seiten-Basisadresse.)

Jeder Eintrag im Seitentabellen-Verzeichnis ist genau 4 byte lang, so daß (in der 4-kbyte-Seite) maximal 1024 Einträge untergebracht werden können. Die Adressierung eines bestimmten Eintrags erfolgt durch die Konkatenation der signifikanten Bits des CR3-Registers mit den höchstwertigen 10 Bits der linearen Adresse, dem *Directory*-Teil, der die Nummer des Eintrags im Seitentabellen-Verzeichnis enthält. Die unteren beiden Bits der Adresse werden auf '0' gesetzt (s. Bild 4.4-4) und zeigen so auf das erste Byte des Eintrags.

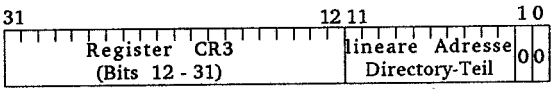


Bild 4.4-4. Selektierung eines Eintrags im Seitentabellen-Verzeichnis

Jeder Eintrag des Seitentabellen-Verzeichnisses (*page directory entry*) weist auf eine bestimmte Seitentabelle und enthält nähere Informationen über den Zustand der Seitentabelle (s. Bild 4.4-5). Auf diese Informationen gehen wir weiter unten ein.

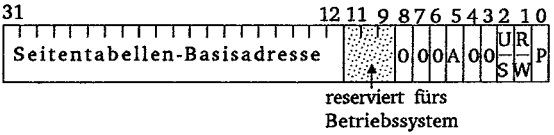


Bild 4.4-5. Eintrag in einem Seitentabellen-Verzeichnis

Die höchstwertigen 20 Bits des Eintrags bestimmen die obersten 20 Bits der Basisadresse der selektierten Seitentabelle. Daran wird der *Table*-Teil der linearen

Adresse (Bits 12-21) angehängt, der die Nummer des Eintrags im Seitentabellen-Verzeichnis bestimmt (s. Bild 4.4-6). Weil jeder Eintrag ebenfalls eine Länge von genau 4 byte aufweist, sind auch bei dieser Basisadresse die beiden niederstwertigen Bits auf '0' gesetzt.

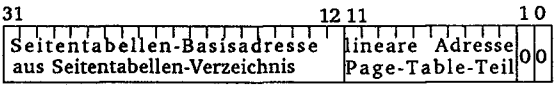


Bild 4.4-6. Selektierung eines Eintrags in der Seitentabelle

Die Seitentabellen

Ein Eintrag in einer **Seitentabelle** (*page table entry*) hat ein Format gemäß Bild 4.4-7.

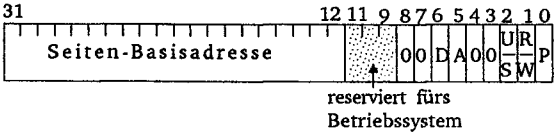


Bild 4.4-7. Eintrag einer Seitentabelle

Mit der 20 bit langen Basisadresse der selektierten Seiten aus der Seitentabelle und dem 12 bit langen *Offset*-Teil der linearen Adresse (Bits 0-11) läßt sich schließlich die 32 bit lange physikalische Adresse des gesuchten Speicherwortes bestimmen (Bild 4.4-8).

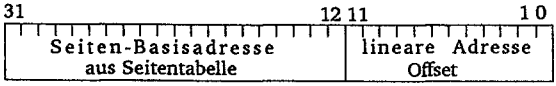


Bild 4.4-8. Berechnung der physikalischen Adresse

Abschließend müssen wir noch die restlichen, bei beiden Tabellentypen identischen Bits der Einträge erklären (vgl. Bild 4.4-5 und 4.4-8). Sie dienen – wie bei den Segment-Deskriptoren – der Implementierung von Schutzmechanismen und unterstützen das Betriebssystem bei der effizienten Durchführung von Speicher-Ersetzungsstrategien.

P (*present bit*) Genau wie bei Segment-Deskriptoren zeigt dieses Bit an, ob die spezifizierte Seitentabelle bzw. Seite im Hauptspeicher vorhanden ist. Wird auf eine nicht vorhandene Seite oder Seitentabelle zugegriffen, dann liegt der bereits erwähnte Seitenfehler (*page fault*) vor, und die gewünschte Seite muß vom Betriebssystem in den Speicher eingelagert werden.

U/S (*user/supervisor bit*) Beim Zugriff auf Speicherseiten wird nur zwischen zwei Privileg-Ebenen unterschieden. Ist das Bit U/S=1 gesetzt, so dürfen alle Prozesse auf diese Seite zugreifen (Benutzer-Modus, *user mode*). Gilt U/S=0, so ist nur Prozessen der Privileg-Ebenen 0-2 (vgl. Abschnitt 4.3.2.1) ein Zugriff auf die Seite erlaubt (Betriebssystem-Modus, *supervisor mode*).

R/W (*read/write bit*) Bei Speicherseiten wird nicht zwischen Code- und Datenseiten unterschieden. Durch das R/W-Bit wird allerdings geregelt, ob im Benutzer-Modus auf eine Seite, die durch U/S=1 gekennzeichnet ist, nur lesender oder auch schreibender Zugriff erfolgen darf. Nur wenn das R/W-Bit gesetzt ist, kann die Seite auch beschrieben werden. Für Seiten, die durch U/S=0 geschützt sind, hat dieses Bit keine Bedeutung; sie dürfen immer gelesen oder beschrieben werden, aber nur im Betriebssystem-Modus.

A (*accessed bit*) Dieses Bit wird vom Prozessor nach jedem Seitenzugriff in der Seitentabelle und dem zugehörigen Eintrag im Seitentabellen-Verzeichnis gesetzt. Mit Hilfe dieses Bits kann das Betriebssystem feststellen, auf welche Seiten schon lange nicht mehr zugegriffen worden ist (vgl. Selbsttestaufgabe S4.4-1.) Dadurch kann es entscheiden, welche Seite als nächste ausgelagert werden kann.

D (*dirty bit*) Dieses Bit wird nur in der Seitentabelle benutzt, um anzuzeigen, daß in der entsprechenden Seite ein Speicherwort verändert worden ist. Der Inhalt sehr vieler Seiten, z.B. solcher, die Programmcode enthalten, wird während ihrer Einlagerung im Arbeitsspeicher nicht verändert. Diese Seiten müssen deshalb bei der Ersetzung durch andere Seiten nicht in den Hintergrundspeicher zurückgeschrieben werden.

Die restlichen drei Bits (Bits 9-11) stehen dem Betriebssystem zur Verfügung. Dort können zusätzliche Informationen über die Seite bzw. Seitentabelle abgespeichert werden, auf die wir hier nicht eingehen wollen.

Selbsttestaufgabe S4.4-1:
Wie läßt sich mit Hilfe des *Accessed Bit* möglichst einfach eine LRU-ähnliche Ersetzungsstrategie implementieren ?

Selbsttestaufgabe S4.4-2:

Bestimmen Sie für den Intel 80386 die physikalische Adresse, wenn die folgende lineare Adresse (in Hexadezimalschreibweise) gegeben ist:

$FFFFF002 .

Das CR3-Register soll den Wert $001FA000 enthalten.

Deuten Sie insbesondere die Tabellen-Einträge:

- Darf das durch die Adresse selektierte Datum verändert werden ?
- Befindet sich die zugehörige Seite im Hauptspeicher ?
- Ist auf die Seite bereits zugegriffen worden ?
- Dürfen auch Prozesse der Privileg-Ebene 3 auf das Datum zugreifen ?

Der Hauptspeicher habe die folgende Belegung (Ausschnitt):

Adresse	Inhalt	Adresse	Inhalt
$00200FFF	0 0		
$00200FFE	3 0	$001FB000	3 1
$00200FFD	1 0	$001FAFFF	0 0
$00200FFC	0 6	$001FAFFE	2 0
$00200FFB	4 0	$001FAFFD	0 0
$00200FFA	A 1	$001FAFFC	2 5
$00200FF9	1 1	$001FAFFB	3 0
$00200FF8	2 3	$001FAFFA	0 4
$00301006	4 5	$001FA004	A 1
$00301005	2 B	$001FA003	0 2
$00301004	0 0	$001FA002	3 0
$00301003	0 5	$001FA001	4 1
$00301002	2 0	$001FA000	0 4
$00301001	3 0		
$00301000	0 2		

Welcher Wert ist in der adressierten Speicherstelle abgelegt?

4.4.1.3 Beschleunigung der Adreßberechnung durch einen Cache

Normalerweise erfolgt die Berechnung einer physikalischen Adresse, wie oben beschrieben, über den Zugriff auf das Seitentabellen-Verzeichnis und die entsprechende Seitentabelle. Um diese Berechnung zu beschleunigen, benutzt der 80386

einen speziellen, sehr schnellen, assoziativen Cache-Speicher, der *Translation Lookaside Buffer* (TLB) genannt wird. Der Prozessor lädt automatisch die 32 zuletzt benutzten Einträge aus dem Seitentabellen-Verzeichnis und der Seitentabelle in den Cache. Bei jeder linearen Adresse, die in eine physikalische umgesetzt werden muß, wird zunächst nachgesehen, ob die durch die höchstwertigen 20 Bits der linearen Adresse spezifizierten Tabellen-Einträge im Cache sind. In diesem Fall, der Treffer (*hit*) genannt wird (vgl. Abschnitt 3.8), werden die oben beschriebenen Tabellen nicht benötigt, die Basisadresse der Seite befindet sich im TLB. Gibt es den gewünschten Eintrag nicht im Cache, so muß mit Hilfe der im Hauptspeicher residierenden Tabellen die physikalische Adresse bestimmt werden (vgl. Bild 4.4-9).

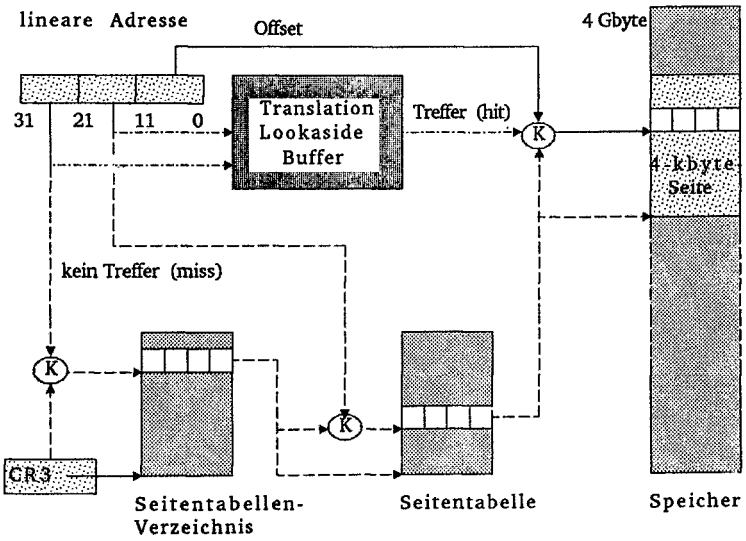


Bild 4.4-9. Benutzung eines Caches zur Beschleunigung der Adreßberechnung

Von der Herstellerfirma wird angegeben, daß es in typischen Anwendungen bei weit über 90% der Seitenzugriffe einen Treffer gibt, d.h. die Adreßberechnung kann über den schnellen Cache erfolgen. Eine hohe Treffer-Rate läßt sich wiederum durch die Lokalitätseigenschaft von Programmen erklären.

4.4.1.4 Vorgehensweise bei Seitenfehlern

Entscheidend für eine effiziente Verwaltung des Seitenspeichers durch das Betriebssystem ist die Häufigkeit, mit der benötigte Seiten vom Hintergrundspeicher in den Hauptspeicher eingelagert werden müssen. Ein Maß dafür ist die soge-

nannte Seitenfehler-Rate (*page fault rate*), d.h. die Anzahl der erforderlichen Seiteneinlagerungen pro Zeiteinheit.

Ist bei einem Seitenzugriff die erforderliche Seitentabelle nicht im Hauptspeicher, d.h. ist im entsprechenden Eintrag des Seitentabellen-Verzeichnisses das P-Bit nicht gesetzt, generiert der Prozessor eine Unterbrechungs-Anforderung (*exception*). Daraufhin muß die Seitentabelle durch das Betriebssystem eingelagert werden. Genauso wird vorgegangen, wenn in der zweiten Stufe der Adreßberechnung die gewünschte Seitennummer nicht in der Seitentabelle vorliegt, die Seite also nicht im Hauptspeicher residiert.

Zur effizienten Durchführung des Ein- und Auslagerns von Seiten stehen dem Prozessor vier Systemregister CR0 bis CR3, die sogenannten *(System) Control Registers* (s. Bild 4.4-10) zur Verfügung.

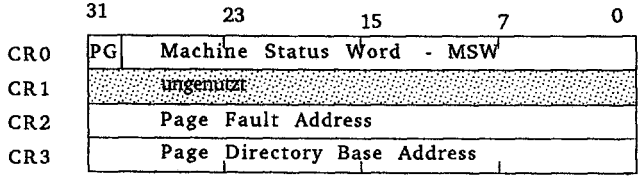


Bild 4.4-10. Systemregister zur Verwaltung der Speicherseiten

Im ersten Register CR0 ist das Maschinenstatuswort (MSW) abgespeichert. Als Erweiterung zum 80286 ist das höchstwertige Bit (*page enable* – PG) dann gesetzt, wenn der Prozessor eine Seitenverwaltung durchführen soll. (Dieses Register kann z.B. im Betriebssystem-Modus durch den Befehl MOVE CR0 geladen werden.) Im Register CR2 wird die lineare Adresse derjenigen Seite abgelegt, die zu einem Seitenfehler geführt hat. Auf dieses Register kann das Betriebssystem zugreifen, um den Seitenfehler zu behandeln. Im Register CR3 ist, wie schon erwähnt, die Basisadresse des Seitentabellen-Verzeichnis des gerade vom Prozessor bearbeiteten Prozesses abgelegt. Bei einem Prozeßwechsel wird automatisch die Basisadresse des neuen Seitentabellen-Verzeichnisses ins Register CR3 eingetragen.

4.4.2 Seitenverwaltungskonzept des MMU-Bausteins NS32382

Die MMU NS32382 der Firma National Semiconductor ist ein spezieller Baustein, der für den 32-bit-Prozessor NS32332 die Speicherverwaltung übernimmt. Dieser Zusammenhang ist schematisch in Bild 4.4-11 dargestellt.

Die CPU reicht eine virtuelle Adresse an die MMU weiter, die dann selbständig die zugehörige physikalische Adresse generiert bzw. eventuell vorhandene Fehler entdeckt. In diesem Abschnitt werden wir uns hauptsächlich damit beschäftigen, wie die Kommunikation zwischen CPU und MMU verläuft und welche

Signale dazu zwischen diesen Bausteinen ausgetauscht werden. Zuvor wollen wir kurz vorstellen, wie die Berechnung physikalischer Adressen erfolgt.

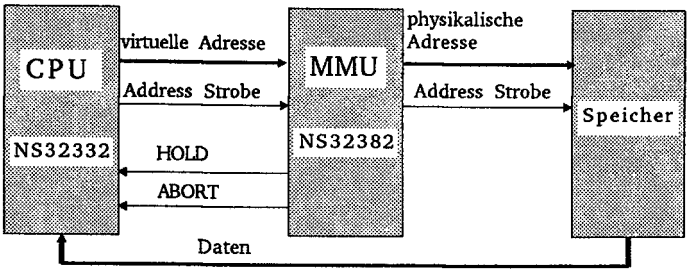


Bild 4.4-11. Zusammenwirken von CPU und MMU zur Berechnung physikalischer Adressen

4.4.2.1 Berechnung physikalischer Adressen aus virtuellen Adressen

Im Gegensatz zum oben vorgestellten Intel-Prozessor 80386 unterstützt die MMU NS32382 ausschließlich eine Seitenverwaltung; eine Aufteilung des Speichers in Segmente unterschiedlicher Größe ist nicht möglich. Deshalb werden virtuelle Adressen – anders als beim 80386 (vgl. Bild 4.4-1) – direkt in physikalische Adressen umgewandelt.

Mit der gegebenen Wortlänge von 32 bit kann ein virtueller Adreßraum von 4 Gbyte benutzt werden. Als Seitengröße wurde auch bei diesem Baustein 4 kbyte gewählt. Die Berechnung der physikalischen Adressen erfolgt durch eine zweistufige Seitentabellen-Verwaltung, die in Bild 4.4-12 dargestellt ist. (Im Unterschied zu Bild 4.4-3 haben wir hier die Konkatenation verschiedener Adreßteile durchgeführt und die dadurch erhaltenen Adressen explizit angegeben.)

Bei dieser MMU besteht eine virtuelle Adresse, genau wie beim Intel 80386, aus zwei 10 bit langen Selektorfeldern und einem 12 bit großen Offset (vgl. Bild 4.4-3). Die benutzten Seitentabellen sind ebenfalls 4 kbyte groß, und die Bildung der physikalischen Adressen erfolgt völlig analog zu den Bildern 4.4-4, 4.4-6 und 4.4-8. Im ersten Byte jedes Tabelleneintrags sind wiederum Informationen über die Zugriffsrechte (*access byte*) abgespeichert. Genau wie bei den Intel-Prozessoren enthält dieses Byte u.a. eine Privileg-Ebene, ein *Dirty Bit* und ein *Accessed Bit*. Jeder Prozeß besitzt wieder sein eigenes Seitentabellen-Verzeichnis, auf das über ein spezielles Basisregister, dem Seitentabellen-Basisregister (STBR) zugegriffen wird.

Um die Berechnung physikalischer Adressen zu beschleunigen, besitzt die MMU NS32382 ebenfalls einen Cache-Speicher, in dem die Tabelleneinträge für die 32 zuletzt benutzten virtuellen Adressen abgelegt sind. Gibt es bei einer Adreßumsetzung einen Treffer, dann sind keine zeitaufwendigen Tabellenzugriffe erforderlich, vgl. Bild 4.4-9. Der Cache wird von der MMU in Anlehnung an das

LRU-Prinzip verwaltet. Nur in wenigen Details (und in der Nomenklatur), auf die wir hier nicht näher eingehen wollen, sind konzeptionelle Unterschiede zur Paging-Einheit des Intel 80386 feststellbar.

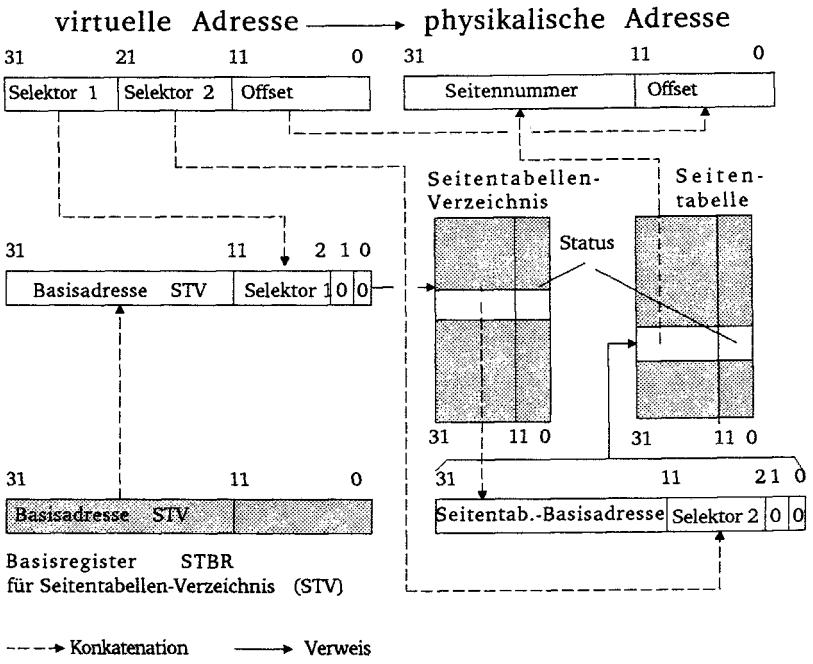


Bild 4.4-12. Berechnung physikalischer Adressen bei der MMU NS32382

4.4.2.2 Kommunikation zwischen CPU und MMU

Als erstes wollen wir die wichtigsten Ein-/Ausgabesignale der MMU zur Steuerung ihrer Kommunikation mit der CPU vorstellen. Ein aus CPU, MMU und Bus-Controller bestehendes System ist in Bild 4.4-13 dargestellt. Der Bus-Controller hat die Aufgabe, aus den Steuer- und Statussignalen der CPU das zeitgerechte Takt- und READY-Signal für die MMU zu generieren. Die meisten Signale sind bereits aus Kapitel 1 bekannt. Der Vollständigkeit halber werden sie hier noch einmal aufgeführt.

Eingangssignale der MMU

$\overline{\text{ADS}}$ (*address strobe*) Der Systembus der CPU überträgt Adressen und Daten im Multiplexbetrieb. Dieses Signal zeigt an, daß sich auf ihm augenblicklich eine (virtuelle) Adresse befindet.

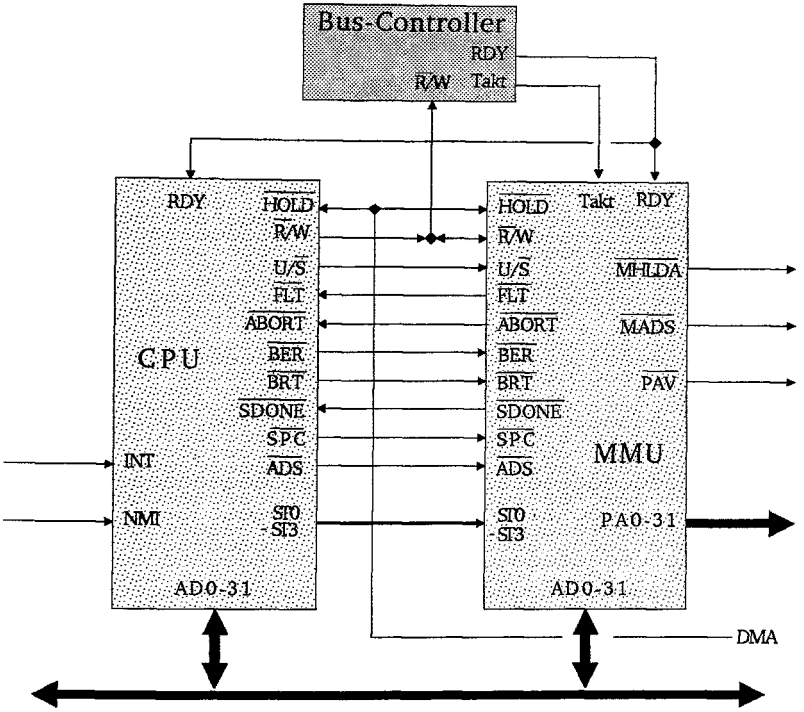


Bild 4.4-13. MMU NS32382 in Verbindung mit CPU und Bus-Controller

RDY *(ready)* Dieses Signal wird vom Bus-Controller auf Wunsch langsamer Komponenten erzeugt und zur Anforderung von Wartezyklen benutzt. Es ist mit dem RDY-Eingang der CPU und der MMU verbunden (vgl. den semi-synchronen Systembus im Abschnitt 1.10).

ST0-ST3 *(status lines)* Die vier Leitungen übertragen die im Abschnitt 1.5 beschriebenen Prozessor-Status-Signale (Funktionscode). Sie definieren den Typ des aktuell durch die CPU ausgeführten Buszyklus.

U/$\overline{S}$ *(user/supervisor)* Dieses Signal wird von der CPU generiert und zeigt an, ob sich die CPU gerade im Benutzer-Modus (*user mode*) oder im Betriebssystem-Modus (*supervisor mode*) befindet. Es wird u.a. zur Implementierung von Schutzmechanismen benutzt.

$\overline{BRT}$ *(bus retry)* Dieses Signal wird von der CPU oder anderen Komponenten gesetzt, wenn der letzte Buszyklus wiederholt werden muß.

$\overline{\text{BER}}$ (*bus error*) $\overline{\text{BER}}$ zeigt an, daß der gerade durchgeführte Buszyklus fehlerhaft war, und dieser Fehler nicht durch eine Wiederholung behoben werden kann.

$\overline{\text{HOLD}}$ (*MMU hold*) Dieses Signal veranlaßt die MMU, den Systembus für einen DMA-Transfer freizugeben. Es entspricht in seiner Funktion dem $\overline{\text{HOLD}}$-Eingang der CPU.

$\overline{\text{SPC}}$ (*slave processor control*) Das $\overline{\text{SPC}}$-Signal wird von der CPU als *Strobe*-Signal benutzt. Mit seiner Hilfe werden die Operationscodes und Daten zur MMU übertragen.

Ausgangssignale der MMU

$\overline{\text{FLT}}$ (*float*) Wenn die MMU auf die Seitentabellen zugreifen will, wird von ihr das $\overline{\text{FLT}}$-Signal gesetzt, damit die CPU den Bus freigibt. D.h. hier handelt es sich um einen speziellen $\overline{\text{HOLD}}$-Ausgang der MMU für die CPU.

PA0-PA31 (*physical address*) Diese Leitungen bilden den Bus, der die physikalische Adresse zum Speicher überträgt.

$\overline{\text{MADS}}$ (*MMU address strobe*) Dieses Signal zeigt an, daß sich eine physikalische Adresse auf dem Bus PA0-PA31 befindet. Es wird von der MMU als Triggersignal für einen Adreßpuffer ausgegeben, wenn die CPU der MMU den Buszugriff eingeräumt hat, insbesondere also für den Zugriff auf die Seitentabellen und die Seitentabellen-Verzeichnisse.

$\overline{\text{PAV}}$ (*physical address valid*) Dieses Signal meldet dem Speicher, daß die Adresse auf dem Bus PA0-PA31 gültig ist.

$\overline{\text{ABORT}}$ Mit diesem Signal wird ein von der CPU initialisierter Buszyklus abgebrochen.

$\overline{\text{MHLDA}}$ (*MMU hold acknowledge*) Dieses Signal zeigt bei einer DMA-Operation an, daß die MMU den Bus freigegeben hat. Es entspricht in seiner Funktion einem $\overline{\text{HLDA}}$ (*hold acknowledge*) der CPU.

$\overline{\text{SDONE}}$ (*slave done*) $\overline{\text{SDONE}}$ wird von der MMU gesetzt, um der CPU die Beendigung einer MMU-Instruktion anzuzeigen.

Ein-/Ausgabesignale der MMU

AD0-AD31 (*address*) Diese Signale bilden den "gemultiplexten" Daten-/Adreßbus. Auf dem Bus werden sowohl die virtuellen Speicheradressen zur MMU als auch die Daten aus dem Speicher zur CPU bzw. in umgekehrter Richtung übertragen.

$\overline{\text{R}}$/W (*read/write*) Das $\overline{\text{R}}$/W-Signal bestimmt während eines Buszyklus die Richtung eines Datentransfers (Lesen: $\overline{\text{R}}$/W=0, Schreiben: $\overline{\text{R}}$/W=1).

4.4.2.3 Durch die CPU initiierte Buszyklen

Ein durch die CPU initiierter Buszyklus benötigt vier Taktzyklen T_1-T_4, vgl. Bild 4.4-14.

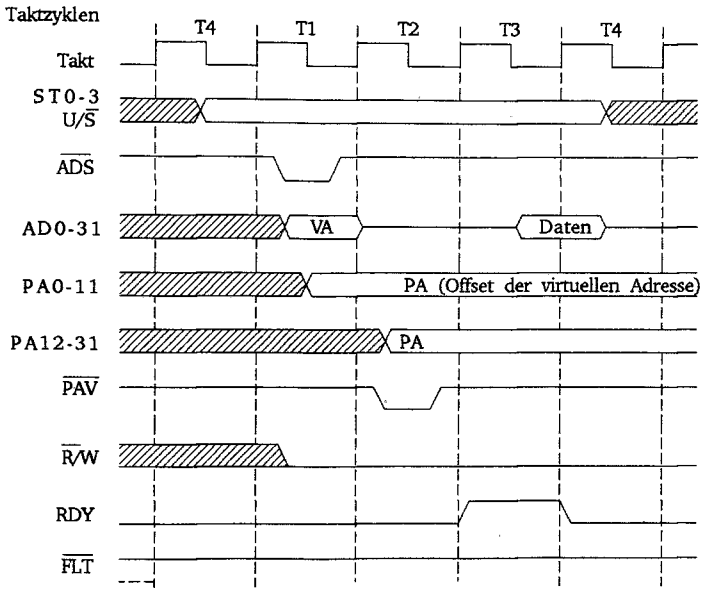


Bild 4.4-14. CPU-Buszyklus eines Lesebefehls bei einem Cache-Treffer

Während des ersten Taktzyklus T_1 legt die CPU die virtuelle Adresse (VA) auf den Bus AD0 - AD31 und generiert das Übernahmesignal $\overline{\text{ADS}}$ (*address strobe*). Die niederstwertigen 12 Bits, d.h. der Offset, werden direkt auf den physikalischen Adreßbus PA0 - PA11 durchgeschaltet. Die restlichen Bits werden von der MMU gelesen, um daraus die Bits PA12 - PA31 zu bestimmen. Wie dies geschieht, werden wir nun beschreiben.

Während des zweiten Taktzyklus T_2 nimmt die CPU die virtuelle Adresse wieder vom Systembus. Nun können drei verschiedene Situationen auftreten.

1. Die virtuelle Adresse befindet sich im Cache, d.h. es wird ein Treffer festgestellt, und es liegt keine Verletzung der Zugriffsrechte vor. In diesem Fall kann die MMU die physikalische Adresse PA12 - PA31 noch innerhalb von T_2 auf den physikalischen Adreßbus legen und dies durch das $\overline{\text{PAV}}$-Signal (*physical address valid*) anzeigen. Bild 4.4-14 zeigt den Zeitverlauf bei einem Cache-Treffer, wenn Daten aus dem Speicher gelesen werden.

2. Die virtuelle Adresse ist im Cache vorhanden, aber die Zugriffsrechte auf die entsprechende Seite werden verletzt. Dann generiert die MMU kein $\overline{\text{PAV}}$-Signal, sondern sendet stattdessen ein $\overline{\text{ABORT}}$-Signal an die CPU, um anzuzeigen, daß die gewünschte Adreßberechnung nicht zulässig ist. Dies ist im Bild 4.4-15 dargestellt.

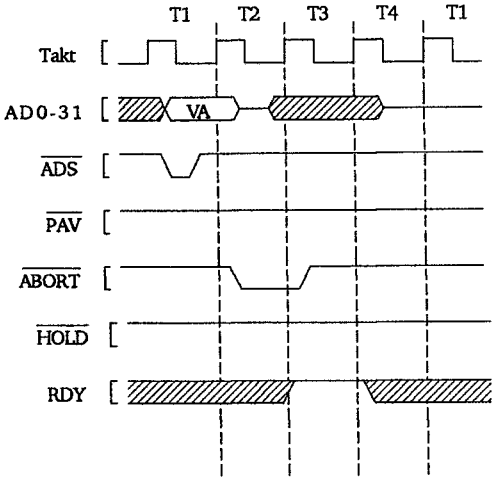


Bild 4.4-15. Abbruch eines Buszyklus wegen einer Verletzung der Zugriffsrechte

3. Die virtuelle Adresse befindet sich nicht im Cache, d.h. es gibt keinen Treffer, und es muß ein Zugriff auf die im Hauptspeicher liegenden Seitentabellen erfolgen. Ein solcher Tabellenzugriff ist auch erforderlich, wenn eine Seite erstmalig verändert wird und ihr *Dirty Bit* noch ungesetzt ist. In diesen Fällen muß die MMU die Buskontrolle übernehmen und meldet diesen Wunsch der CPU mit dem $\overline{\text{FLT}}$-Signal (*float*). Der Ablauf eines solchen durch die MMU initiierten Buszyklus wird im nächsten Unterabschnitt beschrieben (s. Bild 4.4-16).

Wird der Buszyklus nicht wegen einer Verletzung der Zugriffsrechte abgebrochen, so werden die Daten während der Taktzyklen T_3 und T_4 auf dem Systembus zwischen CPU und Speicher übertragen. Dies ist ebenfalls im Bild 4.4-14 dargestellt.

Innerhalb von T_3 wird zusätzlich das vom Bus-Controller generierte RDY-Signal (*ready*) von der MMU überprüft. Ist es in T_3 auf L-Potential, dann werden zusätzliche Wartezyklen durch die Wiederholung von T_3 ausgeführt. Dadurch ist es möglich, den Buszyklus an langsamere Datenübertragungen anzupassen.

4.4.2.4 Durch die MMU initiierte Buszyklen

Sobald ein Zugriff auf die Seitentabellen erfolgen muß, übernimmt die MMU die Buskontrolle. Für die Berechnung der physikalischen Adresse werden wegen der zweistufigen Seitentabellen-Verwaltung zwei zusätzliche Buszyklen benötigt. Bild 4.4-16 zeigt den zeitlichen Ablauf. Darin wird beispielhaft ein Schreibbefehl durchgeführt ($\overline{R}/W=1$).

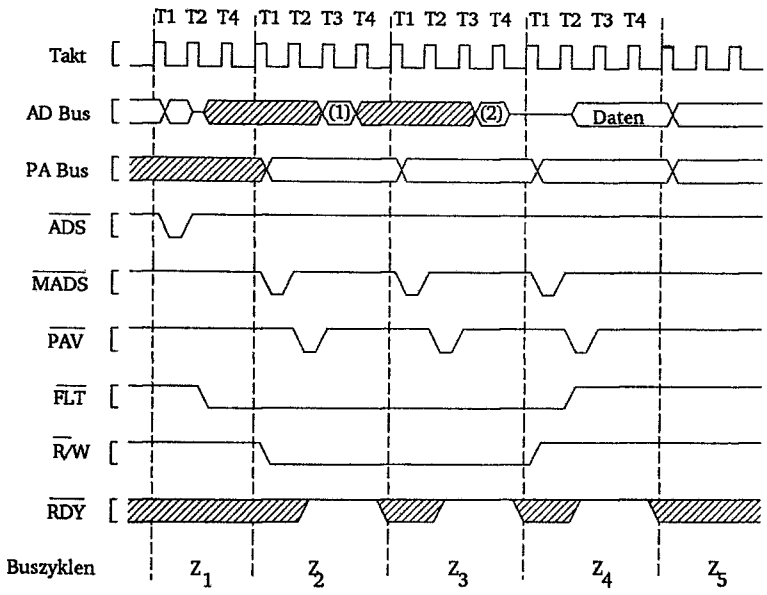


Bild 4.4-16. MMU-initiierter Buszyklus zum Zugriff auf die Seitentabellen

Nachdem im ersten Buszyklus Z_1 die MMU der CPU durch eine negative Flanke des $\overline{FLT}$-Signals (*float*) signalisiert hat, daß sie auf den Bus zugreifen will, wartet sie eine zusätzliche Taktphase (T_4). Im Buszyklus Z_2 wird der gesuchte Eintrag aus dem Seitentabellen-Verzeichnis (1), in Z_3 aus der Seitentabelle (2) gelesen. Im vierten Buszyklus Z_4 kann die MMU in T_1 die physikalische Adresse auf PA0-PA31 legen und dies durch das Signal $\overline{MADS}$ (*MMU address strobe*) anzeigen. In T_2 setzt sie das FLT-Signal zurück und generiert ein $\overline{PAV}$-Signal, damit in T_3 und T_4 mit einem CPU-Buszyklus die eigentliche Datenübertragung, wie oben beschrieben, fortgesetzt werden kann.

Muß in den Seitentabellen die Statusinformation, z.B. das *Dirty Bit* oder das *Accessed Bit*, modifiziert werden, geschieht dies durch zusätzlich eingefügte Schreibzyklen. Sie schließen sich direkt dem zweiten bzw. dritten Buszyklus an.

Wartezyklen

Um verschiedene Übertragungsraten, z.B. von langsamen Peripheriegeräten, zu ermöglichen, können Wartezyklen durch den Bus-Controller erzeugt werden (vgl. Bild 4.4-13). Dies geschieht durch das RDY-Signal (*ready*), das während der Taktzeit T_3 in den Bildern 4.4-14 und 4.4-16 stets auf '1' gesetzt war. Gilt RDY=0, so wird nach T_3 nicht mit T_4 fortgefahren, sondern nochmals ein Takt T_3 eingefügt.

Zeitverhalten bei Busfehlern

Wird bei einer Übertragung ein leichter Busfehler festgestellt, d.h. einer, der durch eine Übertragungs-Wiederholung behoben werden kann, aktiviert die CPU das $\overline{\text{BRT}}$-Signal (*bus retry*). Die MMU prüft $\overline{\text{BRT}}$ während der Taktzeiten T_3 und T_4. Befindet sich $\overline{\text{BRT}}$ dann auf einem L-Pegel, wird der gesamte Buszyklus wiederholt.

Wird ein schwerer Busfehler entdeckt, der durch Wiederholung nicht behoben werden kann, erkennt dies die MMU während der Taktzeit T_4 durch ein gesetztes $\overline{\text{BER}}$-Signal (*bus error*), das von der CPU ausgegeben wird. In diesem Fall wird die virtuelle Adresse, bei der dieser Fehler aufgetreten ist, in einem speziellen Register BEAR (*bus error address register*) abgespeichert und im MMU-Statusregister ein entsprechendes Bit gesetzt. Anschließend wird mit einem $\overline{\text{ABORT}}$-Signal der Buszyklus abgebrochen und die Buskontrolle durch Deaktivieren von $\overline{\text{FLT}}$ an die CPU zurückgegeben.

DMA-Transfer

Durch Setzen der $\overline{\text{HOLD}}$-Leitung kann ein DMA-Gerät die MMU zur Busfreigabe veranlassen. Die MMU erteilt dann durch das $\overline{\text{MHLDA}}$-Signal (*MMU hold acknowledge*) dem DMA-Baustein die Buskontrolle. Wenn der DMA-Controller seinen Buszugriff beendet hat, setzt er das $\overline{\text{HOLD}}$-Signal zurück, und die MMU erhält wieder die Buskontrolle.

4.4.2.5 MMU-Instruktionen

Die MMU NS32382 enthält eine Reihe von Registern, mit deren Hilfe die CPU die Funktionen der MMU steuern kann. Die wichtigsten MMU-Register sind:

PTB0, PTB1 (*page table base register*) Jedes dieser Register enthält die Basisadresse für ein Seitentabellen-Verzeichnis.

BEAR (*bus error address register*) In das BEAR-Register wird die virtuelle Adresse, bei der ein Busfehler aufgetreten ist, abgespeichert (s.o.).

MSR (*memory management status register*) Das MSR enthält wichtige Statusinformationen. Dazu gehören u.a. Informationen darüber,

- ob ein Lese- oder ein Schreibzyklus durchgeführt worden ist,
- ob und an welcher Stelle Zugriffsrechte verletzt worden sind,
- welche Busfehler aufgetreten sind und

– ob die zu berechnende Adresse zu einem im Benutzer-Modus (*user mode*) oder Betriebssystem-Modus (*supervisor mode*) befindlichen Prozeß gehört.

Sämtliche Register können von der CPU durch bestimmte Instruktionen gelesen und verändert werden. Weil die MMU für die CPU als Slave-Prozessor arbeitet, werden diese Befehle *Slave Instructions* genannt. Alle MMU-Instruktionen sind privilegiert, d.h. sie können nur im Betriebssystem-Modus benutzt werden. Es sind vier verschiedene MMU-Befehle möglich:

- SMR (*store MMU register*),
- LMR (*load MMU register*).

Mit diesen Befehlen können die MMU-Register gesetzt bzw. deren Inhalte gelesen werden.

- RDVAL (*validate address for reading*),
- WRVAL (*validate address for writing*).

Mit diesen Instruktionen läßt sich feststellen, ob bei einem Schreib- bzw. Lesezugriff auf eine gegebene Adresse eine Verletzung der Zugriffsrechte auftreten würde. Mit ihnen kann somit vor der Ausführung eines Befehls geprüft werden, ob der gewünschte Befehl zulässig ist, und somit verhindert werden, daß eine auftretende Fehlersituation zu einem Trap führt.

Das Format eines MMU-Befehls ist in Bild 4.4-17 dargestellt.

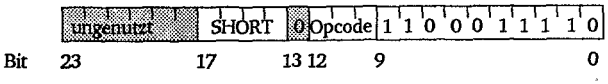


Bild 4.4-17. Format einer MMU-Instruktion

Die niederstwertigen 10 Bits identifizieren den Befehl als eine Slave-Prozessor-Instruktion, die an die MMU gerichtet ist. Im OpCode-Feld wird einer der vier MMU-Befehle spezifiziert. Im 4 bit langen SHORT-Feld wird das vom Befehl angesprochene MMU-Register selektiert, die höchstwertigen Bits 18 - 23 bleiben ungenutzt.

Für die Übertragung eines MMU-Befehls gibt es wiederum ein genaues Protokoll. Hierbei benötigt ein Buszyklus nur die zwei Taktzeiten T_1 und T_4 (vgl. Abschnitt 4.2.2.3), weil keine explizite Adreßberechnung durchgeführt werden muß. Bei jedem dieser Buszyklen werden genau 32 Bits von oder zur MMU übertragen. Das Zeitverhalten ist in den Bildern 4.4-18 und 4.4-19 dargestellt.

Während der Taktzeit T_1 wird von der CPU das *Strobe*-Signal $\overline{SPC}$ (*slave processor control*) aktiviert, d.h. auf L-Pegel gesetzt. Bei einem Lesebefehl übernimmt die CPU zu Beginn von T_4 die auf dem Bus befindlichen Daten (s. Bild 4.4-18).

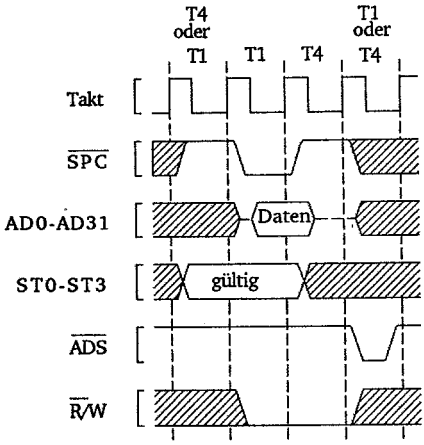


Bild 4.4-18. Buszyklus für eine MMU-Instruktion: CPU liest aus der MMU

Wenn die CPU zur MMU schreiben will (s. Bild 4.4-19), legt sie in derselben Taktzeit die Daten auf den Systembus. Innerhalb von T_4 deaktiviert die CPU dann das Signal $\overline{SPC}$.

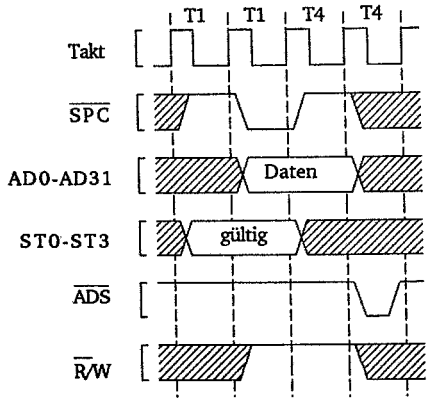


Bild 4.4-19. Buszyklus für eine MMU-Instruktion: CPU schreibt zur MMU

Als Beispiel wollen wir nun noch in Tabelle 4.4-20 die einzelnen Schritte darstellen, die bei einem LMR-Befehl (*load MMU register*) in der CPU und in der MMU stattfinden. Eine wichtige Aufgabe spielt dabei der Funktionscode ST0 - ST3 der CPU, der über spezielle Leitungen (s. Abschnitt 1.5.2) ausgegeben und während eines Slave-Buszyklus von der MMU gelesen wird. Die Schritte für die anderen MMU-Instruktionen sehen ähnlich aus.

Tabelle 4.4-20. Durchführung des LMR-Befehls (*load MMU register*)

die CPU	ST0-ST3	die MMU
legt die MMU-Instruktion auf den Systembus und aktiviert das $\overline{\text{SPC}}$-Signal	1 1 1 1	akzeptiert und dekodiert den Befehl
schickt den Operanden zur MMU und aktiviert $\overline{\text{SPC}}$	1 1 0 1	nimmt den Operanden vom Bus und setzt das angegebene Register
wartet aufs $\overline{\text{SDONE}}$-Signal der MMU	0 0 1 1	aktiviert $\overline{\text{SDONE}}$, um das Befehlsende anzuzeigen

4.5 Schutzmechanismen

Im folgenden werden wir uns mit den weiteren Möglichkeiten moderner Mikroprozessoren zur Unterstützung von Betriebssystemen beschäftigen, speziell zur Realisierung von Schutzmaßnahmen (*protections*) und der Multitasking-Betriebsart. Sobald es um genaue Implementierungen geht, greifen wir auf unseren Beispiel-Prozessor Intel 80286 zurück; gegebenenfalls werden wir auf Unterschiede beim Seitenspeicherkonzept des 80386 hinweisen.

Vom Prozessor werden während der Laufzeit von Programmen eine Reihe von Konsistenzüberprüfungen (*protection checks*) vorgenommen, um nicht erlaubte Speicherzugriffe zu verhindern. Generell wird auf drei Ebenen Schutz gewährleistet:

- Trennung der Systemsoftware, z.B. des Betriebssystems, insbesondere des Ein-/Ausgabe-Subsystems (BIOS – *basic I/O system*), von den Anwendungsprozessen.
- Trennung der Anwendungsprozesse voneinander. Ist dies nicht gewährleistet, könnte ein fehlerhaftes Anwendungsprogramm andere, fehlerfreie Programme beeinflussen.

- Datentyp-Überprüfungen während der Laufzeit. Es muß z.B. gesichert sein, daß nicht versucht wird, Datensegmente als Programme zu interpretieren und auszuführen bzw. Codesegmente zu beschreiben, und daß der Offset einer Adresse nicht außerhalb der Segmentgrenzen führt.

4.5.1 Schutzebenen

Das wichtigste Mittel zur Realisierung von Schutzmechanismen sind die sogenannten Schutz- oder Privileg-Ebenen (PL – *privilege levels*). In vielen µP-Systemen, z.B. beim Motorola 68000 oder beim National Semiconductor 16000, wird (nur) zwischen zwei verschiedenen Privileg-Ebenen, nämlich dem Betriebssystem-Modus (*supervisor mode*) und dem Benutzer-Modus (*user mode*) unterschieden. Dabei darf ein Auftrag im Benutzer-Modus in der Regel keine Daten oder Programme des höherprivilegierten Betriebssystem-Modus benutzen.

Das Konzept der zwei Schutzebenen wurde beim Intel 80286 auf eine vierstufige Hierarchie der Vertrauenswürdigkeit (*hierarchy of trust*) erweitert (vgl. Abschnitt 4.3.2.1). Entsprechend gibt es vier verschiedene Privileg-Ebenen (PL). Die Privileg-Ebene PL=0 entspricht der vertrauenswürdigsten Ebene (*most trusted level*), PL=3 der am wenigsten vertrauenswürdigen Ebene (*least-trusted level*). Diese vier Ebenen ermöglichen eine differenzierte Unterscheidung zwischen den verschiedenen Arten von Code und Daten.

Ein typisches System, das alle vier Privileg-Ebenen ausnutzt, ist in Bild 4.5-1 dargestellt. Dies Bild macht auch deutlich, warum man anstelle von Schutzebenen auch von Schutzringen (*protection rings*) spricht.

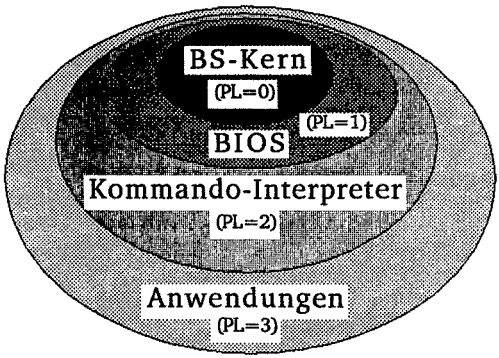


Bild 4.5-1. Beispielsystem mit vier verschiedenen Schutzringen

Die in Bild 4.5-1 dargestellten Systemkomponenten sind:

- Der Betriebssystemkern (BS-Kern, *kernel*) realisiert die wichtigsten Aufgaben des Betriebssystems, wie z.B. die Speicherverwaltung, das Prozeß-Management

und die Interrupt-Behandlung. Er ist der wichtigste Software-Teil des Systems, der von allen Benutzerprogrammen benötigt wird. Deshalb wird er als besonders fehlerfrei und zuverlässig vorausgesetzt.

- Das BIOS (*Basic Input/Output System*) ist ein Teil des Betriebssystem-Kerns und befindet sich resident im Rechner. Es beinhaltet alle standardisierten Grundfunktionen, die für die Ein-/Ausgabe benötigt werden, und macht die darüber liegenden Schichten des Betriebssystems von der aktuellen Rechnerhardware unabhängig.
- Der Kommando-Interpreter (*command line interpreter*) bildet die Benutzerschnittstelle des Betriebssystems. Diese Komponente nimmt die vom Benutzer eingegebenen Befehle an, interpretiert sie und übergibt sie zur Ausführung an das Betriebssystem.
- Die Anwendungsprogramme (*applications*) sind vom Benutzer geschriebene Programme, die normalerweise keine systemweiten Dienste zur Verfügung stellen.

Selbstverständlich müssen nicht immer alle vier Privileg-Ebenen auch wirklich benutzt werden. Bei einem ungeschützten System können alle Segmente derselben Privileg-Ebene (PL=0) angehören. Genauso ist es möglich, nur zwei verschiedene Ebenen zu unterscheiden, beispielsweise den Benutzer-Modus und den Betriebssystem-Modus.

4.5.2 Zugriffsrechte

Zugriffsrechte (*access rights*) garantieren, daß nur unter bestimmten Voraussetzungen auf die im Speicher abgelegten Informationen zugegriffen werden darf. Das angewendete Grundprinzip ist dabei das folgende:

> Jeder Auftrag darf nur auf diejenigen Daten zugreifen, die er wirklich zur Erfüllung seiner Aufgabe benötigt.

In diesem Abschnitt müssen wir klären, wo der Prozessor Privileg-Ebenen benutzt und wie sie zur Realisierung von Schutzmechanismen herangezogen werden.

4.5.2.1 Schutzmaßnahmen bei Segmentverwaltung

Um Schutzmaßnahmen bei einer Segmentverwaltung zu realisieren, werden Schutz- oder Privileg-Ebenen an verschiedenen Stellen eingesetzt. Dabei wird sowohl den Daten- als auch den Codesegmenten eine Privileg-Ebene zugeordnet, die – wie wir im Abschnitt 4.3.2 gezeigt haben – im Segement-Deskriptor spezifiziert wird. Also besitzt jedes Datensegment ein eigenes Privileg, aber auch die Ausführung jedes Prozesses findet auf einer bestimmten Privileg-Ebene statt.

Zugriffsregeln

Bei der Realisierung von Schutzmechanismen muß zwischen einem Zugriff auf Daten und einem Zugriff auf Programmcode unterschieden werden. Dabei gelten

generell die im Bild 4.5-2 für eine vierstufige Vertrauenshierarchie veranschaulichten Regeln für den Zugriffsschutz (*protection rules*):

- Ein Prozeß darf nur auf Daten zugreifen, die höchstens genauso vertrauenswürdig (*trusted*) sind wie er selbst.
- Ein Prozeß darf nur Code benutzen, der mindestens genauso vertrauenswürdig ist wie er selbst.

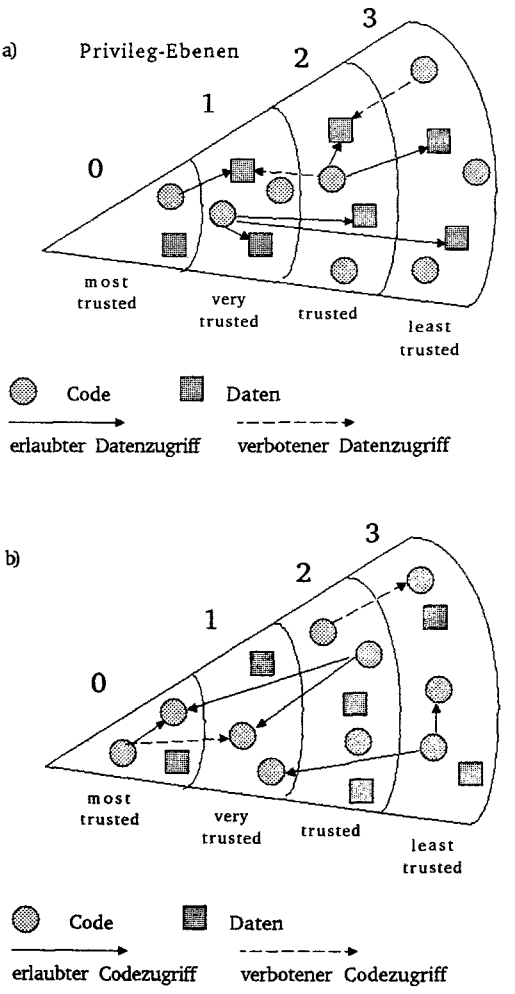


Bild 4.5-2. Regeln für den Zugriffsschutz

Die erste Regel ist insbesondere deshalb notwendig, damit nicht Prozesse mit geringen Privilegien wichtige Daten verändern, insbesondere die Systemtabellen des Betriebssystems, also z.B. die Deskriptor-Tabellen. Die zweite Regel stellt sicher, daß der aufgerufene Code wenigstens den gleichen Qualitäts- und Sicherheitsansprüchen genügt wie der aufrufende. Insbesondere verhindert sie, daß in einer aufgerufenen Prozedur mit niedrigem Privileg vor dem Rücksprung (durch den Befehl RETURN) in die höhere Privileg-Ebene die Rücksprungadresse auf dem Stack manipuliert und dadurch geschützter Code auf der höheren Ebene angesprungen wird.

Fallstudie: Das Schutzkonzept des Intel 80286

Wir wollen jetzt am Beispiel des 80286 konkret aufzeigen, wie Zugriffsregeln implementiert werden können. Dazu werden u.a. die folgenden Privileg-Ebenen benutzt:

Prozeß-Privileg-Ebene (*Current Privilege Level* – CPL)
Jedem gerade ausgeführten Prozeß wird eine Privileg-Ebene zugeordnet, das *Current Privilege Level*. Das CPL ist eine sich dynamische ändernde (zeitabhängige) Größe, die durch die Privileg-Ebene des gerade ausgeführten Codesegments bestimmt wird.

Deskriptor-Privileg-Ebene (*Descriptor Privilege Level* – DPL)
Die Deskriptor-Privileg-Ebene haben wir bereits im Abschnitt 4.3 mehrfach erwähnt. Sie ist im Segment-Deskriptor abgelegt (s. Bild 4.3-7) und legt die Privileg-Ebene eines bestimmten Speicherbereiches (Segmentes) fest.

Verlangte Privileg-Ebene (*Requested Privilege Level* – RPL)
Die letzten beiden Bits eines Segment-Selektors legen die Privileg-Ebene fest, die von einem selektierten Segment erwartet wird (vgl. die Bilder 4.3-2 und 4.3-12). Sie kann vom Betriebssystem im Selektor modifiziert werden. Durch die Wahl von RPL kann es den Prozeß der Privileg-Ebene EPL:=max(RPL,CPL)[2], der sogenannten **effektiven Privileg-Ebene**, zuordnen. Ob durch die Vorgabe des Selektors im Falle eines Codesegments ein Wechsel der Prozeß-Privileg-Ebene oder bei einem Datensegment ein Speicherzugriff erfolgen darf, hängt davon ab, ob für EPL die Regeln für den Zugriffsschutz erfüllt sind, die wir oben beschrieben haben.

Im "Normalfall" wird RPL:=CPL gesetzt. Jedoch kann das Betriebssystem z.B. mit Hilfe der verlangten Privileg-Ebene RPL verschiedene Prozesse (aber auch denselben Prozeß in unterschiedlichen Situationen) beim Zugriffversuch auf ein bestimmtes Segment dynamisch wechselnden, effektiven Privileg-Ebenen zuordnen, ohne daß die DPL im Segment-Deskriptor geändert werden muß. Wichtig ist dies insbesondere dann, wenn ein Selektor (als Parameter) einer Prozedur C' auf höherer Privileg-Ebene übergeben wird und dort ein Datensegment adressiert. Hier wird RPL auf den Wert CPL des übergebenden Prozesses C gesetzt und da-

2) max: Maximum von

durch verhindert, daß C indirekt, quasi durch die "Hintertür" auf ein Datensegment mit höherem Privileg zugreift.

Interpretation der Regeln für den Zugriffsschutz

Zugriff auf Daten

Ein Prozeß mit der Privileg-Ebene CPL kann auf alle Daten zugreifen, die sich in einem Segment befinden, dessen Privileg-Ebene DPL zahlenmäßig mindestens genauso groß ist. Es muß also gelten: CPL≤DPL. Nach der 1. Regel für den Zugriffsschutz ist es also möglich, Datensegmente zu verwenden, die weniger vertrauenswürdig sind, d.h. deren DPL numerisch größer als das aktuelle CPL ist, oder anders gesagt: Daten dürfen nur durch Code verändert werden, der mindestens genauso vertrauenswürdig wie sie selbst ist !

Zugriff auf Code

Ein Prozeß C mit der Prozeß-Privileg-Ebene CPL kann ohne spezielle Voraussetzungen zunächst nur solchen Code C' benutzen (in Form von CALL- oder JUMP-Befehlen), der dieselbe Privileg-Ebene DPL wie er selbst besitzt (vgl. Bild 4.5-3a), d.h. es muß im Deskriptor des Codesegmentes DPL=CPL gelten.

Der Zugriff auf Code mit einem höheren Privileg, d.h. CPL>DPL, ist nur mit der Hilfe einer speziellen Kontrollstruktur, den sogenannten *Call Gates* (s. Abschnitt 4.5.3) oder durch einen Prozeßwechsel möglich (s. Abschnitt 4.6 und Bild 4.5-3b). Ruft z.B. ein Anwendungsprogramm C mit der CPL=3 über ein Call Gate (CG) eine Betriebssystem-Routine C' mit einer DPL=0 auf, so hat der Prozeß die Privileg-Ebene CPL=0, solange die Betriebssystem-Routine abgearbeitet wird. Danach wechselt er wieder automatisch auf die Privileg-Ebene CPL=3. Für die Entscheidung, ob ein Wechsel der Prozeß-Privileg-Ebene erfolgen kann, d.h. CPL:=DPL gesetzt wird, wird die 2. Regel für den Zugriffsschutz angewandt.

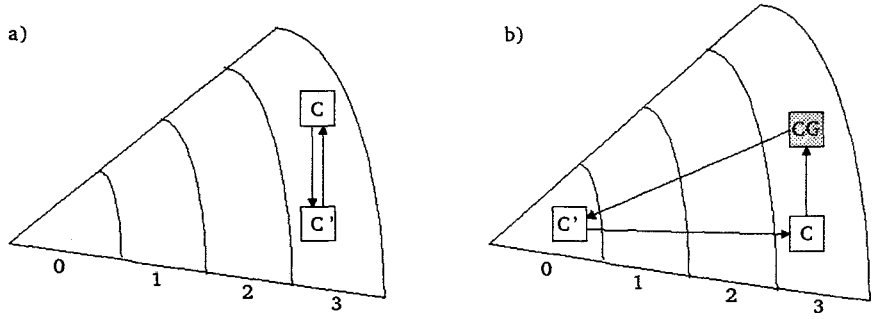


Bild 4.5-3. Zugriff auf ein neues Codesegment;
a) mit derselben Privileg-Ebene, b) mit einer höheren Privileg-Ebene über ein *Call Gate*

Jetzt wollen wir erklären, wie die Zugriffsrechte im einzelnen überprüft werden. Überprüfungen (*protection checks*) werden vom Prozessor in drei Situationen durchgeführt:

- Zugriff auf eine virtuelle Adresse, die einen neuen Selektor enthält,
- Zugriff auf ein Segment,
- Durchführung einer privilegierten Operation.

Überprüfung der Zugriffsrechte beim Zugriff auf einen neuen Selektor

Wie im Abschnitt 4.3.1.2 beschrieben, wird jedesmal, wenn ein neuer Selektor in ein Segmentregister geladen wird, der zugehörige Segment-Deskriptor in den entsprechenden Segment-Deskriptor-Cache geschrieben (vgl. Bild 4.3-2). Auf diesen kann von Anwendungsprogrammen aus nicht explizit zugegriffen werden. Sämtliche Zugriffsüberprüfungen werden vielmehr vom Prozessor selbst hardwaremäßig, d.h. ohne Softwareunterstützung, durchgeführt. Bei jedem neuen Selektor muß geprüft werden, ob das spezifizierte Segment

- im Speicher ist,
- eine für den aktiven Prozeß zulässige Privileg-Ebene hat,
- den richtigen Segmenttyp für die ausgeführte Operation hat.

Wird beispielsweise in einem Befehl eine logische Adresse durch

SELEKTOR:OFFSET

mit einem neuen Selektor definiert, dann werden beim 80286-Prozessor im einzelnen die folgenden Schritte durchgeführt (vgl. Bild 4.5-4):

1. Der Prozessor prüft die Zugriffsrechte über das *Access Byte* im Deskriptor:
   - Ist das Segment im Speicher vorhanden, d.h. ist das *Present Bit* P gesetzt ?
   - Besitzt das Segment den richtigen Segmenttyp ? So muß z.B. das Segmentregister CS auf ein Codesegment verweisen, d.h. im *Access Byte* muß E = 1 gelten.
   - Ist die Privileg-Ebene des Segments nach den oben genannten Zugriffsregeln zulässig ? Gilt also:
     - bei einem Datensegment: CPL≤DPL ?
     - bei einem Codesegment: CPL≥DPL ?
2. Ergibt eine der genannten Prüfungen einen Fehler, dann wird eine Unterbrechung des Programms (*exception*) generiert. (Wie die möglichen Fehler behandelt werden, wird im Abschnitt 4.8 beschrieben.)
3. Wurde kein oder ein behebbarer Fehler festgestellt, so wird im Segmentregister vom Prozessor die erwartete Privileg-Ebene RPL auf die Deskriptor-Privileg-Ebene DPL gesetzt, d.h. RPL: = DPL.
4. Das *Accessed Bit* im Deskriptor wird gesetzt, um festzuhalten, daß ein Segmentzugriff erfolgte.

5. Die Basisadresse, die Größe des Segments sowie seine Zugriffsrechte werden aus dem Segment-Deskriptor in den entsprechenden Segment-Deskriptor-Cache übertragen. Dabei entscheidet der Tabellenindikator TI (*table indicator*) im Selektor, ob der Deskriptor in der LDT oder der GDT zu finden ist.
6. Der Offset der im Befehl benutzten Variablen VAR ist relativ zum Beginn des Segments festgelegt. Zur eigentlichen Berechnung der physikalischen Adresse werden schließlich die Basisadresse aus dem Deskriptor-Cache und der Offset addiert.

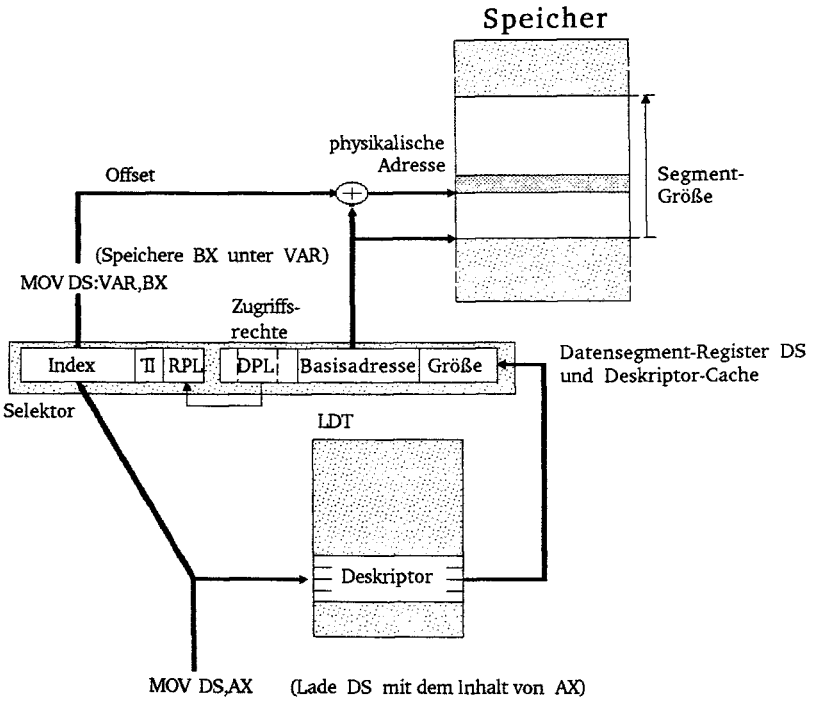


Bild 4.5-4. Berechnung der physikalischen Adresse beim Befehl 'MOV DS,AX' mit neuem Selektor

Überprüfung der Zugriffsrechte bei jedem Segmentzugriff

Weil jeder Code und alle Daten immer in Segmenten mit bestimmten Privileg-Ebenen und Zugriffsrechten gespeichert sind, kann auch bei jedem Segmentzugriff eine Konsistenzüberprüfung stattfinden:

- Lese-/Schreib-Recht:
  Beim Lesen von Daten eines Codesegments muß das R-Bit gesetzt sein. Sollen neue Daten in ein Datensegment geschrieben werden, muß W=1 gelten (vgl. Abschnitt 4.3.2).
- Segmentgröße:
  Zusätzlich wird bei jedem Segmentzugriff überprüft, ob der Offset der virtuellen Adresse innerhalb der durch die Segmentgröße (*limit*) gegebenen Grenzen liegt, d.h. kleiner bzw. bei *Expand-Down*-Segmenten (vgl. Abschnitt 4.3.2) größer als die Segmentgröße ist.

Überprüfung der Zugriffsrechte bei Ausführung einer privilegierten Operation

Bestimmte privilegierte Operationen dürfen nur von Prozessen der Privileg-Ebene PL=0, also z.B. durch Betriebssystem-Prozesse, benutzt werden. Dabei handelt es sich z.B. um solche Operationen, die Systemregister verändern. So kann verhindert werden, daß Systemtabellen durch nicht hinreichend vertrauenswürdige Anwendungsprozesse verfälscht werden. Bei den Intel-Prozessoren ist z.B. der Befehl

LLDT *Load LDTR (lade das lokale Deskriptor-Tabellen-Register (LDTR) mit einem neuen Selektor)*

eine privilegierte Operation.

4.5.2.2 Schutzmaßnahmen bei Seitenverwaltung

Genau wie einem Segment können auch jeder einzelnen Seite bzw. jeder Seitentabelle Zugriffsrechte zugeordnet werden. Auch bei einer Seite kann geprüft werden, ob sie

- sich im Speicher befindet,
- ein Privileg besitzt, für das die Zugriffsregeln dem aktiven Prozeß den Zugriff auf die Seite gestatten,
- beschrieben werden darf.

Beim Intel 80386 ist das Seitenverwaltungskonzept dem Segmentverfahren "aufgesetzt" worden, d.h. zunächst wird mit den Segment-Deskriptoren eine lineare Adresse errechnet, und danach wird mit Hilfe der Seitentabellen die physikalische Adresse bestimmt (vgl. Bild 4.4-1). Die Schutzmaßnahmen werden ebenfalls in zwei Schritten durchgeführt:

- Wie im vorhergehenden Abschnitt beschrieben, werden im ersten Schritt die Schutzanforderungen der Segmente überprüft, die zur Berechnung der linearen Adresse benötigt werden. Wird eine Verletzung der oben genannten Regeln vom Prozessor festgestellt, so wird unverzüglich eine Programm-Unterbrechung (*exception*) generiert.
- Im zweiten Schritt werden die Attribute zum Schutz der Seiten betrachtet. Wie teilweise schon in Abschnitt 4.4 erläutert wurde, werden bei der Berechnung der physikalischen Adresse die folgenden Überprüfungen vorgenommen:

- Befindet sich die gewünschte Seite im Hauptspeicher, d.h. ist das *Present Bit* P gesetzt) ?
- Ist das U/S-Bit (*user/supervisor*) nicht gesetzt, dann muß die Prozeß-Privileg-Ebene CPL≤2 sein.
- Ist beim Beschreiben einer Seite im Benutzer-Modus (U/S=1) das R/W-Bit (*read/write*) gesetzt ?

Diese Überprüfungen finden sowohl beim Zugriff auf ein Seitentabellen-Verzeichnis als auch auf eine Seitentabelle statt. Auch hier wird bei der Verletzung einer Zugriffsregel eine Programm-Unterbrechung (*exception*) generiert. Bei jedem Zugriff wird zusätzlich noch das *Accessed Bit* und gegebenenfalls das *Dirty Bit* gesetzt.

4.5.3 Fallstudie: Kontroll-Transfer beim Intel 80286/80386

In diesem Abschnitt soll erklärt werden, wie auf Code mit einer höheren Privileg-Ebene zugegriffen werden kann. Der Aufruf von Programmcode zwischen verschiedenen Privileg-Ebenen soll zwar prinzipiell möglich sein, aber gleichzeitig nur nach bestimmten Schutzregeln erlaubt sein. Im folgenden werden wir das bei den Intel-Prozessoren realisierte *Call-Gate*-Konzept vorstellen.

Wie bereits bei den Zugriffsregeln erwähnt, ist bei Intel-Prozessoren für einen Prozeß mit der Privileg-Ebene CPL der Zugriff auf Code mit einem höheren Privileg DPL, d.h. DPL<CPL, normalerweise nur mit Hilfe einer speziellen Kontrollstruktur, den sogenannten *Call Gates* möglich.

4.5.3.1 Conforming Code Segment

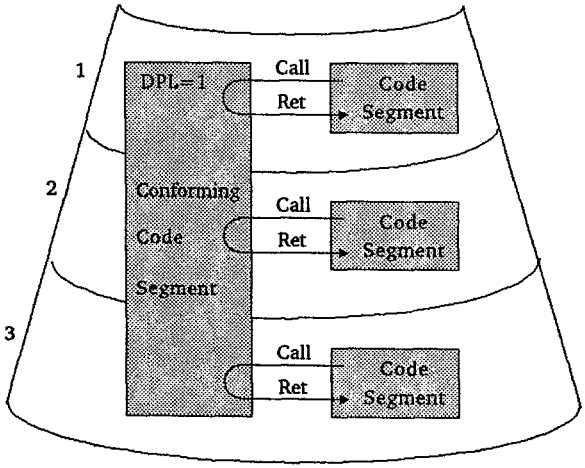


Bild 4.5-5. Aufruf eines Conforming Code Segments beim 80286 von verschiedenen Privileg-Ebenen aus

Beim 80286 kann allerdings beim Zugriff auf höher privilegierten Code auf Call Gates verzichtet werden, wenn im Segment-Deskriptor des aufgerufenen Codesegments das *Conforming Bit* C gesetzt ist (s. Abschnitt 4.3.2). Bild 4.5-5 zeigt den Ablauf des Kontroll-Transfers über ein Conforming Code Segment.

Im Regelfall wird eine aufgerufene Prozedur auf der Privileg-Ebene ausgeführt, die durch den Wert DPL im Deskriptor des Segments, in dem die Prozedur liegt, festgelegt ist. Dies ist bei Conforming Code Segments anders. Hier wird der aufgerufene Code auf derselben Privileg-Ebene ausgeführt wie der aufrufende Prozeß (CPL). (Vorausgesetzt wird natürlich nach den Regeln für den Zugriffsschutz, daß CPL≥DPL ist.) Durch diesen Mechanismus kann das Conforming Code Segment von allen Prozessen benutzt werden, die wenigstens auf derselben Privileg-Ebene arbeiten, und nimmt während seiner Ausführung immer die Privileg-Ebene des aufrufenden Codes an. Es erfolgt somit für den aufrufenden Prozeß kein Privileg-Wechsel.

4.5.3.2 Call Gates (*control transfer*)

Call Gates (CG) regeln die Ausführung von Programmcode, der sich auf einer höheren (d.h. zahlenmäßig kleineren) Privileg-Ebene DPL befindet als derjenigen des aktuell ausgeführten Prozesses. Es gilt also CPL>DPL. Gehört das Codesegment, in das durch das Call Gate gewechselt werden soll, hingegen zu einer niedrigeren Privileg-Ebene DPL, d.h. CPL<DPL, so wird eine Programm-Unterbrechung (*exception*) generiert.

Im Gegensatz zur Verwendung von Conforming Code Segments findet bei Call Gates ein Wechsel der Privileg-Ebene des aktuellen Prozesses statt, d.h. es wird CPL:=DPL gesetzt.

Wechsel der Privileg-Ebene

Ein Wechsel der Privileg-Ebene mit Hilfe eines Call Gates kann durch eine Befehlsfolge

```
CALL FAR <SELEKTOR:OFFSET>      {Unterprogramm-Aufruf}
....                            {Unterprogramm}
....
RET                             {Rücksprung}
```

geschehen, wenn die virtuelle Adresse einen Selektor enthält, der nicht direkt auf den Deskriptor eines neuen Codesegmentes, sondern auf ein Call Gate zeigt. Über das Call Gate erfolgt dann der kontrollierte Zugriff auf den höher privilegierten Code. Der Offset des Sprungziels beim CALL FAR hat in diesem Fall keine Bedeutung.

Der kontrollierte Zugriff auf höher privilegierten Code mit Hilfe von Call Gates ist aus den folgenden Gründen sinnvoll:

- Einerseits benötigen Anwendungsprozesse Zugriff auf Betriebssystem-Dienste.
- Andererseits muß der Zugriff von Anwendungsprozessen auf das Betriebssystem kontrolliert werden, um Schutz zu gewährleisten.

Ein Call Gate ist wiederum ein spezieller, 8 byte langer Deskriptor mit dem in Bild 4.5-6 dargestellten Format, der entweder in der globalen (GDT) oder der lokalen Deskriptor-Tabelle (LDT) eingetragen ist.

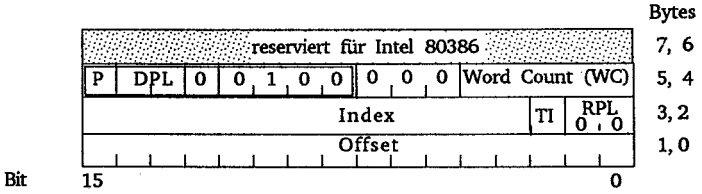


Bild 4.5-6. Call-Gate-Deskriptor beim 80286

Das (doppelt umrahmte) *Access Byte*, das die Zugriffsrechte spezifiziert, unterscheidet einen Call-Gate-Deskriptor von einem Segment-Deskriptor. Ganz allgemein ist das zurückgesetzte Bit 4 (S-Bit) eine Kennzeichnung dafür, daß es sich nicht um einen Segment-, sondern einen Kontroll-Deskriptor handelt. (In den nächsten Abschnitten werden Sie weitere Kontroll-Deskriptoren kennenlernen.)

Der aus Index und Tabellenindikator TI bestehende Selektor (mit RPL=00, vgl. Bild 4.3-12) verweist auf einen Segment-Deskriptor in der LDT bzw. GDT. Der Segment-Deskriptor enthält insbesondere die Basisadresse des Programmcodes. Der Offset gibt die Differenz der Startadresse (*entry point*) des Programms zu dieser Basisadresse an.

Stack-Verwaltung

Für jede der maximal vier Privileg-Ebenen innerhalb eines Prozesses gibt es einen eigenen Stack. Dies ist erforderlich, weil sonst keine vollständige Trennung der verschiedenen Privileg-Ebenen möglich wäre. Beim Aufruf eines Call Gates werden automatisch der Stackpointer sowie der Befehlszähler der durch den CALL-Befehl unterbrochenen Ebene auf dem Stack der neuen Ebene gesichert. Zusätzlich ist es möglich, Parameter vom Stack der aufrufenden Privileg-Ebene auf den Stack der aufgerufenen (höheren) Privileg-Ebene zu kopieren. Der 5 bit lange *Word-Count*-Teil (WC) legt die Anzahl der zu kopierenden Parameter fest (s. Bild 4.5-6). Die Parameterübergabe mit Hilfe von Call Gates wird im Bild 4.5-7 graphisch dargestellt. Dabei ruft ein Prozeß der Ebene PL=3 ein Codesegment der Ebene PL=0 auf.

Beim Aufruf eines Call Gates werden im einzelnen die folgenden Schritte durchgeführt:

- Es wird geprüft, ob die effektive Privileg-Ebene EPL:=max(CPL,RPL) des aufrufenden Prozesses höher als die Deskriptor-Privileg-Ebene DPL des Call Gates oder wenigstens ihr gleich ist, d.h. ob EPL $\leq$ (DPL des Call Gates) ist.

- Das aufgerufene Codesegment (*target*) muß höher oder wenigstens genau so hoch privilegiert sein als/wie der aktuelle Prozeß, d.h. es muß gelten: (DPL des Targets) ≤ CPL.
- Der Stackpointer SS:SP wird auf den neuen Stack gelegt (SS: Stack Segment, SP: Stackpointer).
- Vom Stack des aufrufenden Codesegments werden WC Wörter auf den Stack der neuen, aufgerufenen Privileg-Ebene kopiert.
- Der Befehlszähler CS:IP des durch den CALL-Befehl unterbrochenen Codes wird auf den neuen Stack gelegt (CS: Code Segment, IP: Instruction Pointer).

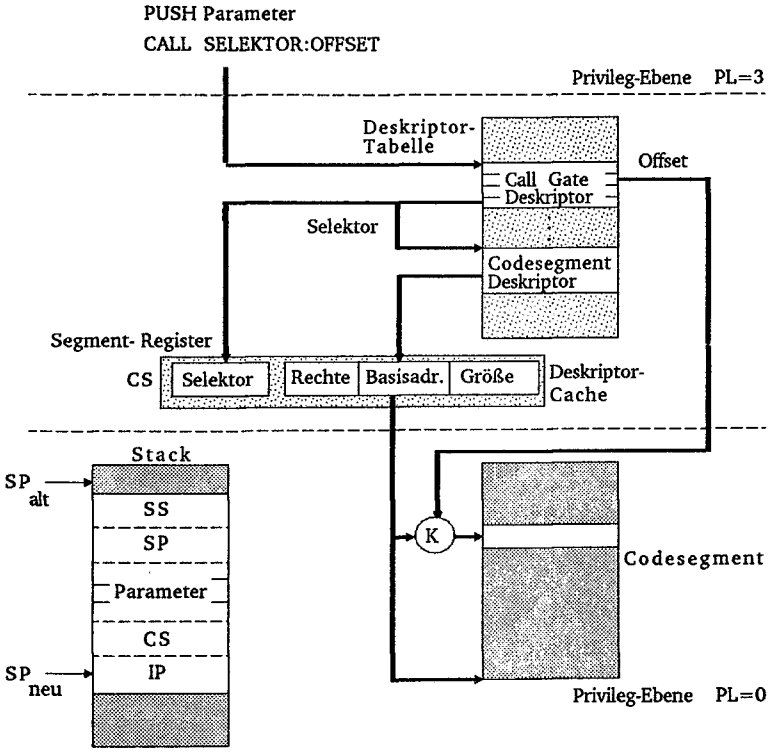


Bild 4.5-7. Parameterübergabe und Stackverwaltung beim Aufruf von Call Gates

Rückkehr ins aufrufende Programm

Die Rückkehr ins aufrufende Programm geschieht durch den Befehl

RET <Anzahl> .

Dabei bezeichnet <Anzahl> die Zahl der vom Stack zu nehmenden Bytes. Durch diesen Befehl geschieht automatisch die Stack-Bereinigung, d.h. es wird die gleiche Anzahl von Bytes vom Stack genommen, wie beim Aufruf des Call Gates bzw. bei der Ausführung seines Programmcodes dort abgelegt wurden.

4.6 Prozeßwechsel

4.6.1 Einleitung

Eine weitere zentrale Aufgabe des Betriebssystems (neben der Speicherverwaltung) besteht darin, Prozesse (*tasks*) zu verwalten (s. Abschnitt 4.1). Besondere Bedeutung besitzen Multitasking-Systeme, in denen es mehrere Prozesse gibt, die abwechselnd vom Prozessor bearbeitet werden (*process multiplexing, process switching, task switching*). Dabei muß ein schneller Prozeßwechsel durch die Hardware unterstützt werden. (Er dauert z.B. beim Intel 80286 zwischen 18 und 20 µs.) Schutzmechanismen sorgen wiederum für einen kontrollierten Prozeßwechsel.

Bei (Multitasking-)Systemen ist es wünschenswert, daß ein Prozeßwechsel nicht nur nach der vollständigen Abarbeitung eines Prozesses erfolgt. Aus Effizienzgründen (vgl. Abschnitt 4.1) ist es vielmehr erforderlich, einen Prozeßwechsel vorzunehmen, wenn irgendein Ereignis eine (Weiter-)Bearbeitung des gerade aktiven Prozesses durch die CPU verhindert. Dazu gehört z.B. das Fehlen einer benötigten Speicherseite, die erst noch vom Hintergrundspeicher eingelagert werden muß. In dieser Situation muß der aktive Prozeß unterbrochen werden, damit die CPU nicht "arbeitslos" (*idle*) wird, sondern mit einem anderen Prozeß fortfahren kann. Der unterbrochene Prozeß geht in den Zustand "blockiert" über (s. Bild 4.1-4). Um die Ausführung dieses Prozesses zu einem späteren Zeitpunkt fortsetzen zu können, ist es erforderlich, sich seinen Zustand zu merken. Unter dem Zustand eines Prozesses versteht man den gesamten Kontext, in dem er sich bei seiner Unterbrechung befindet. Zu diesem Kontext gehören:

- die Inhalte sämtlicher allgemeiner Register,
- der Inhalt des Statusregisters,
- der Wert des Stackpointers,
- der Stand des Befehlszählers,
- sonstige Systemregister-Inhalte.

Kurzum, es müssen sämtliche Informationen über den unterbrochenenen Prozeß abgespeichert werden, die es zum Zeitpunkt seiner Wiederaufnahme ermöglichen, den Prozessor in denselben Zustand zu versetzen, wie er zum Zeitpunkt der Prozeßunterbrechung vorlag. Dieser Prozeß-Kontext wird in einer speziellen Datenstruktur, die oft **Prozeß-Kontroll-Block** (*Process Control Block*) oder auch *Task Control Block* genannt wird, abgespeichert.

4.6.2 Fallstudie: Prozeßverwaltung beim Intel 80286

4.6.2.1 Das Task State Segment

In den Unterlagen der Firma Intel wird statt "Prozeß" (*process*) stets der Begriff *Task* verwendet, den wir deshalb in dieser Fallstudie ebenfalls häufig benutzen werden. Hier ist der Prozeß-Kontroll-Block wiederum ein spezielles Segment, er wird deshalb *Task State Segment* (TSS) genannt. Das TSS eines Prozesses umfaßt beim Intel 80286 mindestens 43 (16-bit-)Wörter und hat den in Bild 4.6-1 dargestellten Aufbau.

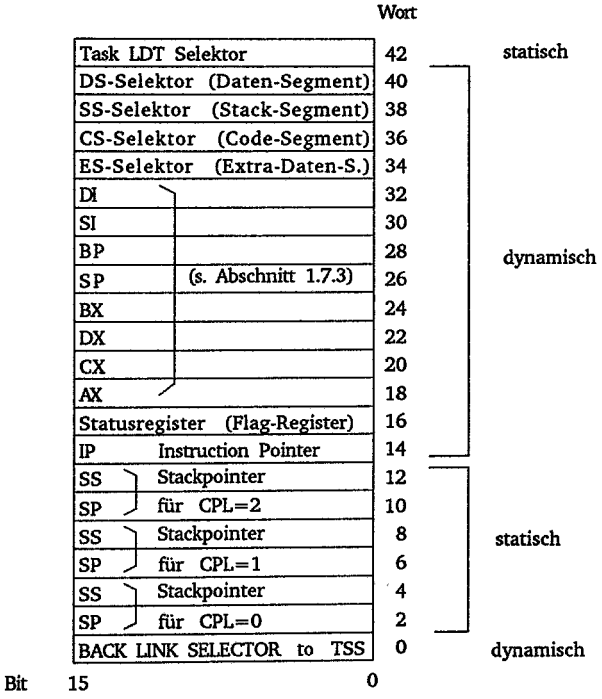


Bild 4.6-1. Der Prozeß-Kontroll-Block beim Intel 80286

Jeder Prozeß-Kontroll-Block besteht zunächst aus einem statischen Teil, der sich während der Prozeßausführung nicht ändert. Dazu gehören:

- der Selektor für die lokale Deskriptor-Tabelle (LDT) des Prozesses und
- die Initialisierungswerte SS:SP der Stackpointer für die Privileg-Ebenen PL=0 bis PL=2.

Der dynamische Teil des TSS umfaßt alle Informationen, die sich während der Laufzeit ändern können. Dazu gehören:

- sämtliche Inhalte der allgemeinen Register,
- die Selektoren in den vier Segmentregistern CS, SS, DS, ES,
- der Inhalt des Statusregisters,
- der Befehlszeiger IP (*instruction pointer*) und
- ein *Back Link Selector* (s.u.).

(Beim Intel 80386 kommt u.a. noch das Systemregister CR3 mit der Basisadresse des Seitentabellen-Verzeichnisses (*page directory*) hinzu.)

Der *Back Link Selector* dient dazu, die Abarbeitung von sogenannten verschachtelten Prozessen (*nested tasks*) zu ermöglichen. Dieser Fall liegt dann vor, wenn nach Beendigung eines Prozesses wieder in den Prozeß zurückgesprungen werden soll, der den Prozeß aufgerufen hatte. Der Back Link Selector verweist dann auf den Prozeß-Kontroll-Block TSS dieses Prozesses. Verschachtelte Prozesse werden daran erkannt, daß im Statusregister EFR (*extended flag register*, vgl. Abschnitt 1.7.3) das NT-Bit (*nested task*) gesetzt ist.

Zusätzlich ist es möglich, daß das Betriebssystem in den Prozeß-Kontroll-Block noch weitere notwendige Informationen hinzufügt, wie z.B. Prioritäten, Speicherplatzbedarf, usw.

Selbsttestaufgabe S4.6-1:
Wo befindet sich im Prozeß-Kontroll-Block TSS der Stackpointer für die Privileg-Ebene PL=3, wenn: a) der zugehörige Prozeß auf der Privileg-Ebene 2 gestartet worden ist ? b) der zugehörige Prozeß auf Privileg-Ebene 2 in den Zustand "blockiert" wechselt ?

4.6.2.2 Der TSS-Deskriptor

Um einen Prozeß-Kontroll-Block (*Task State Segment*) zu beschreiben, wird ein spezieller Deskriptor, der TSS-Deskriptor benutzt, der das in Bild 4.6-2 gezeigte Format aufweist.

Der TSS-Deskriptor enthält, wie alle anderen Deskriptoren, im (doppelt umrahmten) *Access Byte*, das *Present Bit* P und die DPL-Bits, die seine Prozeß-Privileg-Ebene angeben, sowie die spezifische Kennung '000B1' zu seiner Identifikation. Das B-Bit innerhalb dieser Kennung ist gesetzt, wenn der Prozeß gerade bearbeitet wird (*busy*), andernfalls gilt B=0. Des weiteren sind die Basisadresse und die Größe (*limit*) des Prozeß-Kontroll-Blocks abgespeichert TSS. Aus der Be-

schreibung des TSS (s.o.) folgt, daß die Größe stets über \$002A ($=42_{10}$) liegen muß, andernfalls wird eine Exception generiert. Jeder TSS-Deskriptor muß ständig zugreifbar sein, weshalb er in der GDT abgespeichert sein muß.

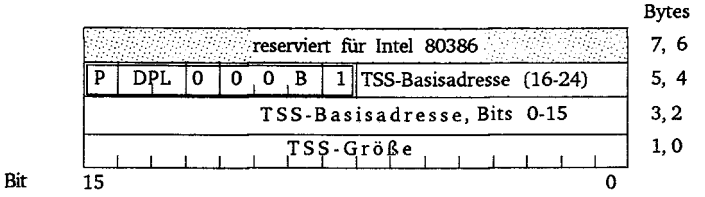


Bild 4.6-2. TSS-Deskriptor

Der Zugriff auf einen Prozeß-Kontroll-Block geschieht über ein spezielles Register, das sogenannte *Task Register* TR, das den entsprechenden Selektor für die GDT enthält. In einem (für das Programm unsichtbaren Teil) des Task Registers werden die Größe des Prozeß-Kontroll-Blocks und seine Basisadresse aus dem selektierten TSS-Deskriptor abgespeichert. Das Task Register verweist stets auf den gerade von der CPU ausgeführten Prozeß. Die Zusammenhänge zwischen Task Register, TSS-Deskriptor und Prozeß-Kontroll-Block TSS werden in Bild 4.6-3 veranschaulicht.

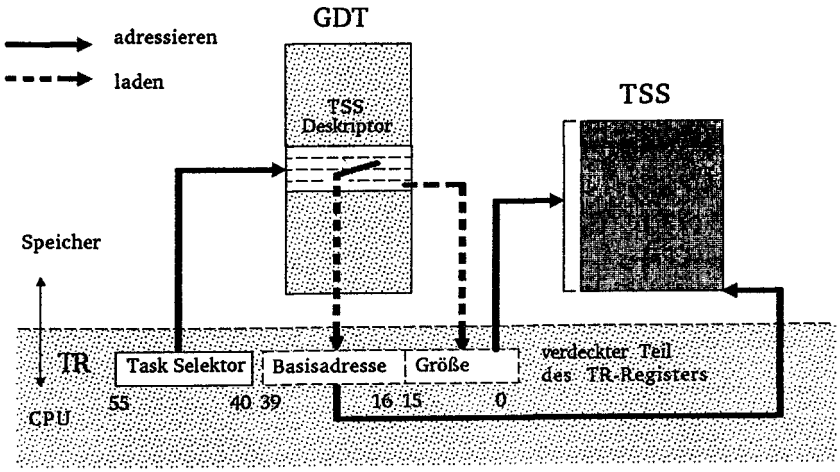


Bild 4.6-3. Adressierung von Prozessen

4.6.2.3 Ursachen für einen Prozeßwechsel

Ein Umschalten zwischen verschiedenen Prozessen (*task switch*) kann alternativ bei der Durchführung eines der beiden Befehle

```
JMP     <Selektor:Offset>   oder
CALL    <Selektor:Offset>
```

erfolgen, wenn der Selektor auf einen TSS-Deskriptor verweist. (Der Offset wird wiederum nicht benötigt). In diesem Fall werden vom Prozessor die folgenden Schritte durchgeführt:

- Alle Register-Inhalte des gerade bearbeiteten Prozesses werden in den Prozeß-Kontroll-Block TSS geladen.
- Das Task Register TR wird mit dem Selektor des neuen TSS-Deskriptors sowie mit der Basisadresse und der Größe des TSS aus diesem Deskriptor geladen.
- Der Registersatz des Prozessors wird komplett neu mit den Registerinhalten geladen, die im neuen, durch den TSS-Deskriptor adressierten Prozeß-Kontroll-Block gespeichert sind. U.a. zeigt dann auch das LDT-Register auf den Deskriptor der neuen lokalen Deskriptor-Tabelle (LDT).

Ein Prozeßwechsel von Task A nach Task B ist in Bild 4.6-4 dargestellt.

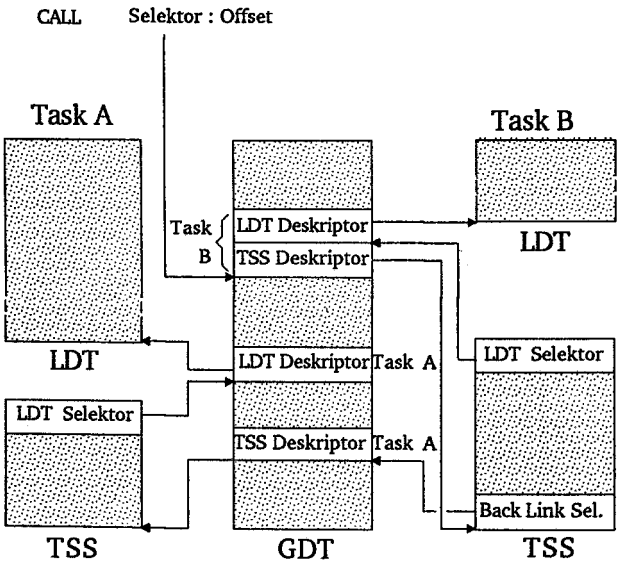


Bild 4.6-4. Prozeßwechsel von Task A nach Task B

4.6.2.4 Zugriffsschutz beim Prozeßwechsel

Auch bei einem Prozeßwechsel werden Schutzmechanismen verwendet. Jeder TSS-Deskriptor muß in der GDT residieren, so daß von jedem Prozeß aus darauf zugegriffen werden kann. Um trotzdem eine unkontrollierte Prozeß-Benutzung zu verhindern, ist normalerweise die Deskriptor-Privileg-Ebene DPL innerhalb jedes TSS-Deskriptors auf 0 gesetzt (vgl. Bild 4.6-2). Dadurch kann ein Prozeßwechsel nur durch einen Prozeß durchgeführt werden, der ebenfalls das Privileg 0 besitzt. Dies dürfte im allgemeinen ein zum Betriebssystem gehöriger Prozeß sein. (Normalerweise ist der *Dispatcher,* der weiter unten ausführlicher beschrieben wird, für das Umschalten auf einen anderen Prozeß zuständig.)

Eine andere Möglichkeit des kontrollierten Zugriffs auf Prozesse ist wiederum durch spezielle *Task Gates* gegeben. In diesem Fall weist bei einem CALL- oder JUMP-Befehl der Selektor-Teil der Adresse nicht direkt auf einen TSS-Deskriptor, sondern auf einen *Task Gate Descriptor*, der eigentlich nur eine bestimmte Deskriptor-Privileg-Ebene DPL sowie den gewünschten TSS-Selektor enthält. Er besitzt daher das in Bild 4.6-5 gezeigte Aussehen.

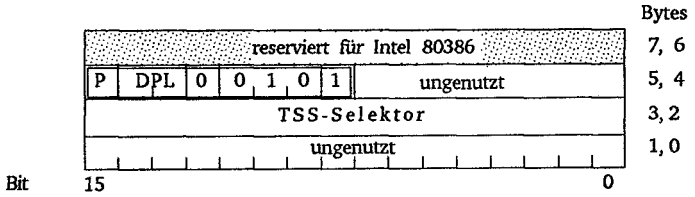


Bild 4.6-5. Aufbau des Task-Gate-Deskriptors

Ein Task-Gate-Deskriptor kann auch in den LDTs verschiedener Prozesse abgelegt sein. Dadurch wird erreicht, daß ein bestimmter Prozeß nur von denjenigen Prozessen benutzt werden kann, in deren LDT der entsprechende Task-Gate-Deskriptor abgelegt ist. Diese Prozesse dürfen dann u.U. auch eine Privileg-Ebene besitzen, die zahlenmäßig größer als 0 ist, also nicht zur höchsten Privileg-Ebene gehören. Einen Prozeßwechsel mit Hilfe eines Task Gates veranschaulicht abschließend Bild 4.6-6.

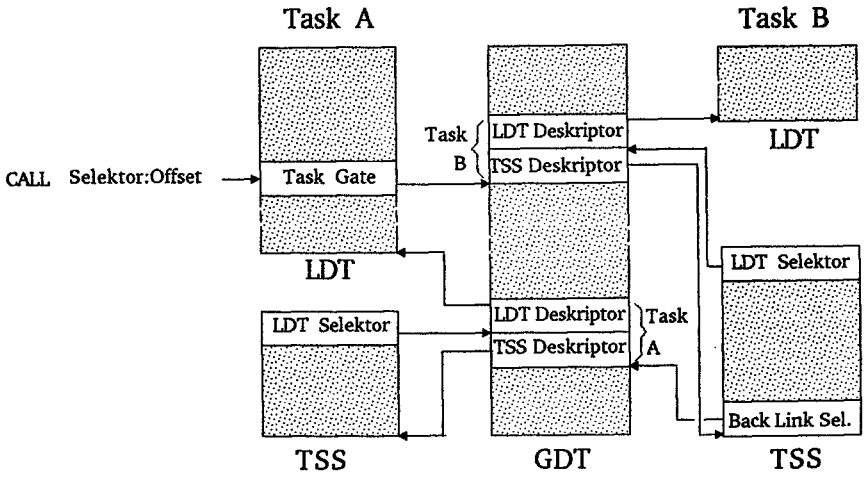


Bild 4.6-6. Prozeßwechsel von Task A nach Task B mit Hilfe eines Task Gates

4.6.2.5 Der Dispatcher

Abschließend wollen wir noch einige Anmerkungen zum Wechseln von Tasks machen. Wie bereits gesagt, ist die Prozeß-Verwaltung eine der zentralen Aufgaben eines Betriebssystems. Der Betriebssystemteil, der festlegt, welcher Prozeß Zugang zum Prozessor erhält, wird *Dispatcher* genannt. Der Dispatcher kann (wenigstens) die in Abschnitt 4.1 genannten Prozeß-Zustände "aktiv", "bereit" und "blockiert" unterscheiden.

Jedesmal wenn ein Prozeß in den Zustand "blockiert" übergeht, kann der Prozessor von diesem nicht mehr genutzt werden. Deshalb kann die CPU in dieser Situation vom Dispatcher einem anderen bereiten Prozeß zugeteilt werden. Dadurch bleibt die CPU beschäftigt (*busy*) und das System wird effizienter genutzt. Sobald das Ereignis eingetroffen ist, auf das der Prozeß im Zustand "blockiert" gewartet hat, kann er wieder vom Prozessor fortgeführt werden, d.h. er geht in den Zustand "bereit" über.

Der Dispatcher muß alle Prozeßzustände verwalten und entscheiden, wann ein Prozeß, der sich im "bereit"-Zustand befindet, der CPU zugeteilt wird. Je nach Aufgabe und Anforderungen des Systems, z.B. Prozeßsteuerung, Dialog-Betrieb, usw. muß der Dispatcher dabei nach verschiedenen Strategien verfahren. Auf diese kann im Rahmen dieses Buches nicht eingegangen werden.

4.7 Kommunikation zwischen Prozessen

In fast allen Systemen gibt es Programmcode und Daten, die von mehreren Prozessen benutzt werden. Beispiele dafür sind das Betriebssystem, aber auch Editoren oder Compiler. Oft ist es auch sinnvoll, daß Prozesse miteinander kommunizieren, d.h. Daten gemeinsam benutzen bzw. unter einander austauschen können (*data sharing*).

Der Wunsch nach Kommunikation zwischen Prozessen steht in gewissem Gegensatz zur Forderung nach Speicherschutz; denn mit den in Kapitel 4.5 vorgestellten Maßnahmen sollte ja gerade erreicht werden, daß die Speicherbereiche der Prozesse streng voneinander getrennt sind. In diesem Abschnitt wollen wir uns damit beschäftigen, welche Möglichkeiten es in Mikroprozessor-Systemen dennoch für den gemeinsamen Zugriff auf Daten durch verschiedene Prozesse gibt.

Mit den uns inzwischen bekannten Konzepten der Speicherverwaltung lassen sich drei unterschiedliche Möglichkeiten des gemeinsamen Gebrauchs von Daten realisieren. Alle drei Ansätze erlauben, daß mehrere Prozesse sowohl auf gemeinsame Segmente als auch auf gemeinsame Seiten zugreifen können. Die Ansätze unterscheiden sich wesentlich dadurch, wie eng die Prozesse aneinander gekoppelt sind, sowie durch den gewährten Schutz.

4.7.1 Kommunikation beim Segmentierungsverfahren

Data Sharing mit Hilfe der GDT

Die einfachste Möglichkeit, mehreren Prozessen den Zugriff auf gemeinsame Segmente (*shared segments*) zu ermöglichen, ist es, die entsprechenden Segment-Deskriptoren in einen globalen, allen Prozessen zugänglichen Adreßbereich abzuspeichern. Bei den Intel-Prozessoren müssen dazu die entsprechenden Deskriptoren in der globalen Deskriptor Tabelle (GDT) abgelegt werden. Der Vorteil dieses Verfahrens ist, daß ohne besonderen Aufwand Änderungen bei den Segmenten, z.B. eine Modifikation der Segmentgröße, vorgenommen werden können, weil es im Gegensatz zu den im folgenden beschriebenen Verfahren nur einen Deskriptor pro Segment gibt.

Dies wird dadurch erkauft, daß alle Prozesse im System auf den Deskriptor und damit auch auf das Segment zugreifen können, falls sie die Zugriffsregeln erfüllen. Oftmals ist es aber wünschenswert, daß nur diejenigen Prozesse eine Zugriffsmöglichkeit erhalten, die das gemeinsame Segment auch wirklich benötigen, d.h. nur eine Teilmenge aller Prozesse im System soll ein Zugriffsrecht erhalten.

Data Sharing mit einer gemeinsamen LDT

Falls mehrere Prozesse fast nur gemeinsame Segmente (*shared segments*) besitzen, ist es sinnvoll, ihnen dieselbe lokale Deskriptor-Tabelle (LDT) zuzuordnen. Dies kann geschehen, indem man den LDT-Selektor des gemeinsamen LDT-Deskriptors in den Prozeß-Kontroll-Block (TSS) aller beteiligten Prozesse lädt (vgl. Bild 4.7-1).

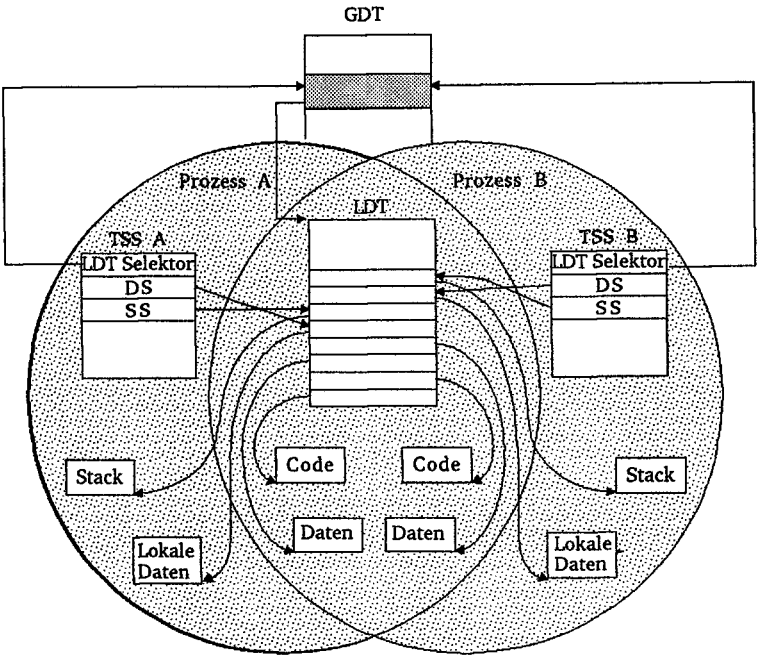


Bild 4.7-1. Data Sharing mit einer gemeinsamen LDT

Der Nachteil dieses Vorgehens liegt auf der Hand: Auf sämtliche Segmente können alle beteiligten Prozesse zugreifen. Es ist hier nicht möglich, daß in differenzierter Weise nur auf einzelne Segmente gemeinsam zugegriffen werden kann. Dieser Nachteil wird beim dritten Verfahren umgangen.

Data Sharing durch Aliasing

Die Methode des *Aliasing* wird in Bild 4.7-2 veranschaulicht. Die beiden Prozesse A und B benutzen ein gemeinsames Datensegment, indem in der LDT jedes Prozesses ein Segment-Deskriptor vorhanden ist, der auf das gemeinsame Datensegment verweist. Es gibt also für dieses Segment mehrere Versionen des zugehörigen Segment-Deskriptors in den LDTs verschiedener Prozesse.

Der große Vorteil dieses Verfahrens liegt in seiner Flexibilität. Ein einzelnes Segment kann von beliebig vielen Prozessen gemeinsam benutzt werden, indem jeder Prozeß für dieses Segment seinen eigenen Deskriptor in seiner LDT hat. Dabei müssen die verschiedenen Versionen der Segment-Deskriptoren nicht identisch sein. Es ist möglich, verschiedenen Prozessen unterschiedliche Zugriffsrechte einzuräumen. So kann ein gemeinsam benutztes Datensegment für Prozeß A nur lesbar, jedoch für einen Prozeß B auch veränderbar (beschreibbar) sein.

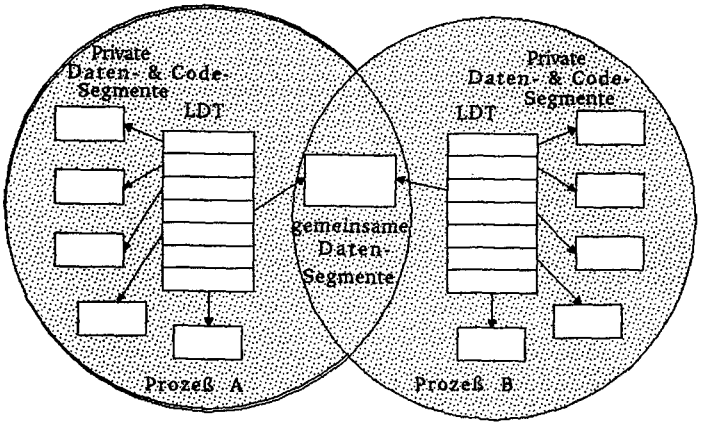


Bild 4.7-2. Data Sharing durch Aliasing

Dieses mächtige Konzept wird durch den Nachteil erkauft, daß der Prozessor selbst für die Konsistenz der verschiedenen Deskriptor-Versionen nicht sorgen kann. Ändern sich also Eigenschaften des Segments, z.B. die Zugriffsrechte oder die Basisadresse bei der Einlagerung in den Hauptspeicher, so müssen sämtliche Alias-Deskriptoren entsprechend modifiziert werden. Diese Aufgabe muß vom Betriebssystem übernommen werden. Dazu muß es sich merken, zu welchen Deskriptoren es Alias-Versionen gibt und in welchen LDTs diese abgespeichert sind.

4.7.2 Kommunikation beim Seitenwechselverfahren

Dieselben Möglichkeiten der Kommunikation, nun aber über gemeinsame Seiten (*shared pages*), bestehen selbstverständlich auch beim Seitenwechselverfahren. In Analogie zu den oben genannten Verfahren gibt es die im folgenden beschriebenen Möglichkeiten.

Data Sharing über eine gemeinsames Seitentabellen-Verzeichnis

Es ist für zwei oder mehrere Prozesse möglich, ein gemeinsames Seitentabellen-Verzeichnis (*page directory*) zu verwalten. In diesem Fall kann auf alle Seitentabellen und sämtliche Seiten gemeinsam zugegriffen werden.

Data Sharing über eine gemeinsame Seitentabelle

Mindestens zwei Prozesse besitzen eine gemeinsame Seitentabelle, d.h. es gibt in beiden Seitentabellen-Verzeichnissen einen Eintrag, der auf dieselbe Seitentabelle weist. Es existieren also zwei Alias-Versionen der Seitentabelle in zwei verschiedenen Seitentabellen-Verzeichnissen. Auch hier können die Einträge in den Verzeichnissen durchaus unterschiedliche Zugriffsrechte zulassen.

Data Sharing über eine gemeinsame Seite

Schließlich gibt es noch die Möglichkeit, eine einzelne Seite gemeinsam zu nutzen. In diesem Fall gibt es mehrere Alias-Versionen für einen Eintrag in einer Seitentabelle.

Genau wie beim segmentierten Speicher muß die Verwaltung der Alias-Tabelleneinträge durch das Betriebssytem unterstützt werden. Dies ist bei der Seitenverwaltung u.U. sehr viel kritischer als bei der Segmentverwaltung. Die bei den Intel-Prozessoren 4 kbyte großen Seiten werden sehr viel häufiger aus dem Hauptspeicher aus- und eingelagert als die im Mittel erheblich größeren Segmente. Bei jeder Einlagerung ändert sich aber normalerweise die Seiten-Basisadresse, so daß relativ häufig sämtliche Alias-Tabelleneinträge modifiziert werden müssen. Deshalb ist es oft sinnvoller, eine Seitentabelle, deren Attribute sich nicht so schnell ändern, gemeinsam zu nutzen als einzelne Seiten.

4.8 Ausnahmebehandlung

Im Abschnitt 1.12 wurden je nach Art des aufgetretenen Ereignisses die folgenden Ausnahme-Situationen (*exceptions*) unterschieden:

- *Interrupts*, die durch externe Ereignisse über die INTR- oder NMI-Leitung generiert werden.
- Durch den INT-Befehl erzeugte *Software-Interrupts*.
- *Traps*, die auftreten, wenn beim Abarbeiten eines Befehls ein Fehler auftritt.

In diesem Abschnitt werden wir beispielhaft die Behandlung der Ausnahme-Situationen durch die Prozessoren 80286/80386 der Firma Intel vorstellen. Dazu zunächst ein Hinweis: Im Gegensatz zum Abschnitt 1.12 wird in den Unterlagen der Firma Intel der Begriff *Interrupt* synonym zum Begriff *Exception* benutzt.

Die Behandlung von Interrupts und Exceptions ist eine spezielle Form eines Kontroll-Transfers. In diesem Abschnitt sollen die durch die MMU generierten Ausnahme-Situationen vorgestellt werden, und wir wollen erläutern, welche Möglichkeiten der Ausnahme-Behandlung es unter Berücksichtigung des Schutzkonzeptes gibt.

4.8.1 Interrupt-Deskriptor-Tabelle

Jeder Interrupt bzw. jeder Trap erfordert seine spezielle Behandlung. Bei Intel-Prozessoren wird deshalb jeder Ausnahme-Situation eine Vektornummer (*exception vector*) zwischen 0 und 255 zugeordnet. Einige dieser Nummern sind bereits vom Hersteller vergeben, andere sind vom Benutzer frei zuzuordnen (vgl. Abschnitt 1.12).

Die Verbindung zwischen dieser Nummer und der ihr zugeordneten Ausnahme-Behandlungsroutine (*exception handler, interrupt handler*) liefert wiederum eine spezielle, im System nur einmal vorhandene Tabelle. Diese wird von Intel In-

terrupt-Deskriptor-Tabelle (IDT) genannt. Wie alle anderen System-Tabellen besteht auch sie aus einer Liste von Deskriptoren, die hier auf die jeweils erforderliche Behandlungsroutine verweisen. Der Zugriff auf die IDT erfolgt ebenfalls durch ein besonderes System-Register, das IDT-Register (IDTR, vgl. Bild 4.8-1). Das IDTR enthält die Basisadresse und die Größe der IDT. Es wird normalerweise vom Betriebssystem mit Hilfe der privilegierten Instruktion LIDT (*load IDT*) geladen.

Ein Zugriff auf die IDT erfolgt nur, wenn eine Ausnahme-Situation aufgetreten ist. Jeder Eintrag in der IDT ist, wie in allen anderen Tabellen, ein 8 byte langer Deskriptor, der einen kontrollierten Übergang (*gate*) zur Ausnahme-Behandlungsroutine darstellt. Je nach Art der Ausnahme-Behandlung werden bei Intel drei verschiedene Arten von *Gates* unterschieden:

- *Task Gates*,
- *Interrupt Gates*,
- *Trap Gates*.

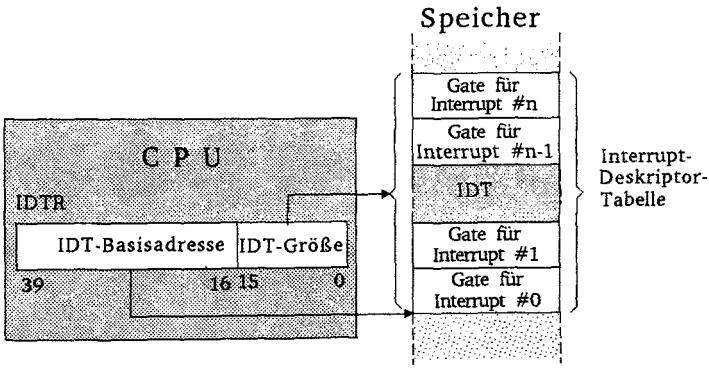


Bild 4.8-1. Interrupt-Deskriptor-Tabelle und IDT-Register beim Intel 80386

Jedes Gate in der IDT besitzt eine eigene Deskriptor-Privileg-Ebene DPL, die festlegt, welches Privileg erforderlich ist, um die Exception- oder Interrupt-Behandlungsroutine aufzurufen. Wie bei den Call Gates muß die Privileg-Ebene des aktuell ausgeführten Prozesses CPL zahlenmäßig größer oder gleich dieser DPL sein. Bei der Ausnahme-Behandlung geschieht ein Wechsel der Privileg-Ebene, wenn für das Codesegment, in dem die Behandlungsroutine abgelegt ist, DPL<CPL gilt. Es gelten dieselben Regeln wie für einen Privilegwechsel über ein Call Gate (vgl. Abschnitt 4.5).

4.8.2 Prozeßorientierte Ausnahme-Behandlung

Eine Möglichkeit der Ausnahme-Behandlung besteht darin, sie durch einen eigenen Prozeß durchführen zu lassen (*task-based handler*). In diesem Fall ist der durch die Exception-Nummer (*exception vector*) spezifizierte Deskriptor in der IDT ein Task Gate (s. Bild 4.8-2). Wie jedes Task Gate verweist dies wiederum auf einen Prozeß-Kontroll-Block TSS, der – wie in Abschnitt 4.6 beschrieben – einen neuen Prozeß, hier zur Unterbrechungs-Behandlung, initialisiert. Bei einer solchen Form der Ausnahme-Behandlung wird – wie bei jedem Prozeßwechsel – automatisch der gesamte, aktuelle Kontext des unterbrochenen Prozesses in seinem TSS gesichert, und das Unterbrechungsprogramm läuft in einem eigenen, neuen Kontext ab. Somit sind der unterbrochene Prozeß (im Beispiel Task A) und die Ausnahme-Behandlung (Interruptroutine) logisch vollständig voneinander getrennt. Um nach der Beendigung eines unterbrochenen Prozesses wieder zu ihm zurückkehren zu können, ist jeder Prozeß zur Behandlung einer Ausnahme-Situation (*exception task*) ein verschachtelter Prozeß (*nested task*, s. Abschnitt 4.6), d.h. im Prozeß-Kontroll-Block der Unterbrechungsroutine ist (als *Back Link Selector*) der TSS-Selektor des unterbrochenen Prozesses eingetragen.

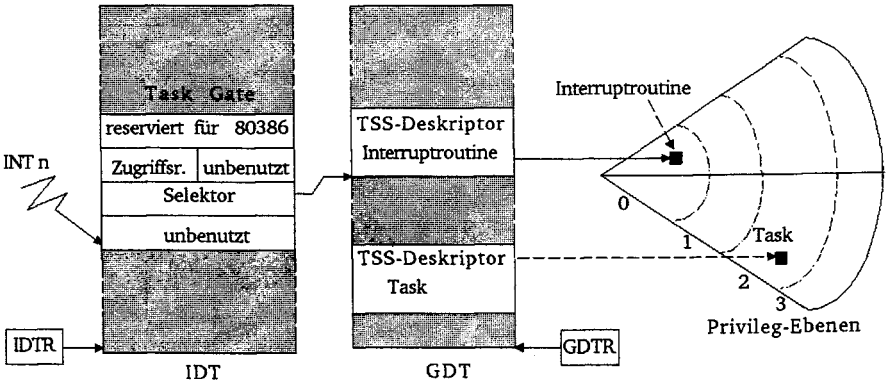


Bild 4.8-2. Prozeß-orientierte Ausnahme-Behandlung

4.8.3 Prozedurorientierte Ausnahme-Behandlung

Zeigt der Exception-Vektor in der IDT auf ein *Trap Gate* oder ein *Interrupt Gate*, dann erfolgt die Ausnahme-Behandlung innerhalb des unterbrochenen Prozesses. Es handelt sich daher um eine prozedurorientierte Ausnahme-Behandlung, in der der Kontext des unterbrochenen Prozesses erhalten bleibt. Im Bild 4.8-3 ist ein *Trap/Interrupt Gate Descriptor* dargestellt.

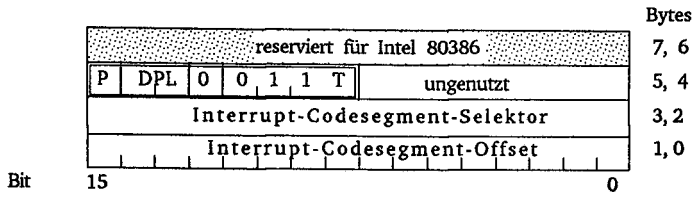


Bild 4.8-3. Trap/Interrupt Gate Descriptor

Ein Interrupt Gate bzw. Trap Gate spezifiziert eine Prozedur, d.h. es verweist mit einem Selektor und einem Offset auf ein Codesegment, und es besitzt fast dieselbe Struktur wie ein Call Gate (s. Bild 4.8-3). Der Prozessor führt einen solchen Prozeduraufruf auch genauso wie einen Call-Gate-Aufruf durch.

Um wieder an die Stelle zurückzukehren, an der der Prozeß unterbrochen wurde, muß jede durch ein Trap Gate bzw. Interrupt Gate initialisierte Ausnahme-Behandlungsroutine durch einen IRET-Befehl (*return from interrupt*) abgeschlossen werden.

Zur Unterscheidung der beiden Gate-Typen ist bei einem Trap Gate im *Access Byte* des Deskriptors das T-Bit gesetzt, bei einem Interrupt Gate nicht. Im Gegensatz zum Call Gate fehlt im Deskriptor lediglich das *Word-Count*-Feld, so daß keine Parameterübergabe des unterbrochenen Codes möglich ist.

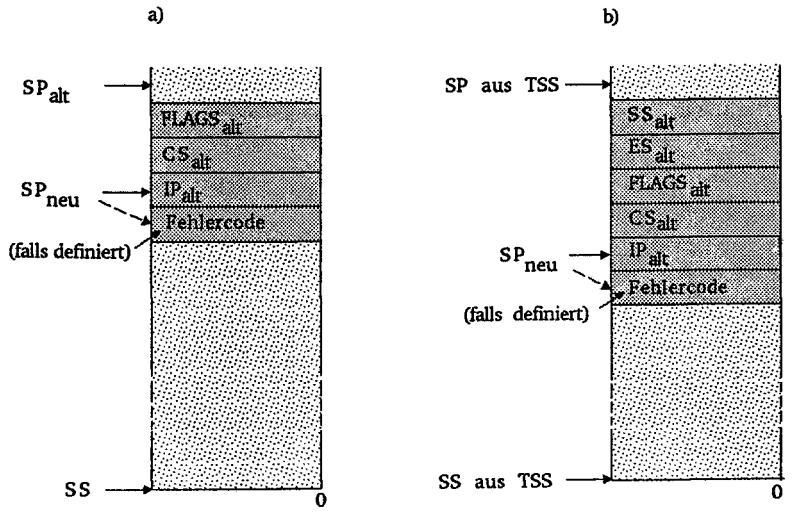


Bild 4.8-4. Belegung des Stacks nach einer prozedurorientierten Ausnahme-Behandlung beim Intel 80286; a) ohne Privileg-Wechsel, b) mit Privileg-Wechsel

Wird durch eine Ausnahme-Situation ein Trap Gate oder ein Interrupt Gate in der IDT selektiert, sichert der Prozessor – wie bei einem Call-Gate-Aufruf – das Statusregister und den Befehlszähler CS:IP (*Code Segment:Instruction Pointer*) auf dem Stack des Interrupt-Programms. Erfolgt ein Privileg-Wechsel, wird zusätzlich noch der alte Stackpointer gesichert. Die Belegung des Stacks nach einer prozedurorientierten Ausnahme-Behandlung ist für diese beiden Fälle in Bild 4.8-4 (s.o.) dargestellt. Die im Bild erwähnten Fehlercodes (*error codes*) werden weiter unten erläutert.

Der Unterschied zwischen einem Trap Gate und einem Interrupt Gate besteht darin, daß im ersten Fall das *Interrupt Enabled Flag* IF[3)] im Steuerregister der CPU gesetzt wird, im zweiten nicht (vgl. Abschnitt 1.7). Dadurch kann die Behandlungsroutine eines Trap Gates nicht ihrerseits durch eine andere Ausnahme-Situation unterbrochen werden. (Unterbrechungen durch NMI-Interrupts sind allerdings weiterhin möglich.) Bei einem Interrupt Gate wird das IF-Bit nicht verändert, so daß je nach seinem aktuellen Zustand Unterbrechungen der Behandlungsroutine möglich sind oder nicht.

Vergleich zwischen prozeß- und prozedurorientierter Ausnahme-Behandlung

Zur Behandlung von internen Unterbrechungen (*traps*) ist oft eine prozedurorientierte Ausnahmeroutine sinnvoll, weil dort der *Interrupt Handler* Zugriff auf sämtliche Daten des unterbrochenen Prozesses besitzt. Ein Beispiel ist ein *Page Fault Handler*, der auf die Seitentabelle des unterbrochenen Prozesses zugreifen muß.

Externe Unterbrechungen (*interrupts*) haben normalerweise keinen Bezug zu dem unterbrochenen Prozeß. Deshalb ist es sinnvoll, sie durch einen eigenen Prozeß zu behandeln, also durch eine prozeßorientierte Ausnahmeroutine. Allerdings dauert ein Prozeßwechsel länger als ein Prozeduraufruf.

4.8.4 Fallstudie: Behandlung der Traps beim Intel 80386

Sobald der Prozessor beim Abarbeiten eines Befehls einen Fehler entdeckt, generiert er einen für diesen Fehler spezifischen Trap. Zur Trap-Behandlung gibt es verschiedene Möglichkeiten zu reagieren:

- Der Fehler kann behoben und danach die abgebrochene Aktion wiederholt werden. Dieses Vorgehen ist für viele Fehler sinnvoll. Wird z.B. festgestellt, daß sich eine benötigte Seite nicht im Hauptspeicher befindet, so kann sie durch den *Page Fault Handler* eingelagert werden, und danach kann der unterbrochene Prozeß mit dem jetzt erfolgreichen Seitenzugriff fortfahren.
- Der Prozeß muß abgebrochen werden. Dies ist bei den Fehlern erforderlich, die nicht behoben werden können, z.B. einer Division durch 0.

3) bei Intel nicht IE abgekürzt !

Als Beispiel wollen wir jetzt die von der MMU im Intel 80386 erzeugten Traps vorstellen. Diese Unterbrechungen finden bei der Speicherverwaltung, z.B. wegen einer Verletzung des Privilegs oder der Zugriffsrechte, statt.

- Segmentüberlauf (*trap* 9)
  Trap 9 wird vom Prozessor generiert, wenn bei der Adreßberechnung die im Deskriptor angegebene Segmentgröße (*limit*) überschritten wird. Dieser Trap kann prozedurorientiert behandelt werden. Allerdings macht es bei diesem Fehler keinen Sinn, die abgebrochene Adreßberechnung zu wiederholen.
- Ungültiges TSS (*trap* 10)
  Dieser Trap wird ausgeführt, wenn bei einem Prozeßwechsel der neue Prozeß-Kontroll-Block (TSS) "ungültig" ist. Ein TSS kann aus sehr vielen Gründen ungültig sein: z.B. kann die Segmentgröße zu klein sein, der Inhalt eines Systemregisters (z.B. LDT-Selektor, Segmentregister CS, usw.) kann fehlerhaft sein, oder die Privileg-Ebene DPL erlaubt keinen Prozeßwechsel. Bei diesem Trap wird ein Fehlercode (*error code*, s.u.) auf den Stack gelegt.
- Segment nicht im Hauptspeicher (*segment fault, trap* 11)
  Dieser Trap wird generiert, wenn beim Laden eines neuen Segments oder beim Zugriff auf ein Gate das *Present Bit* P nicht gesetzt ist. Auch hier wird ein Fehlercode, der den Selektor des Deskriptors enthält, auf den Stack gelegt. Zur Behandlung dieser Unterbrechung wird das fehlende Segment in den Hauptspeicher eingelagert. Anschließend kann der unterbrochene Prozeß fortgesetzt werden.
- Fehler im Schutzkonzept (*general protection fault, trap* 13)
  Alle Fehler, die durch den beim 80386 implementierten Schutzmechanismus erkannt werden und nicht durch einen anderen der genannten Traps behandelt werden, generieren diesen Trap. Darunter fällt das Verletzen der Privileg-Regeln, das Schreiben in ein Segment, auf das nur lesender Zugriff erlaubt ist (*read-only segment*), usw. Normalerweise können diese Fehler nicht behoben werden, so daß der betroffene Prozeß abgebrochen werden muß.
- Seitenfehler (*trap* 14)
  Bei einem Seitenfehler erzeugt der 80386 Prozessor automatisch einen Trap mit der Nummer 14 und speichert die lineare Adresse, bei der ein Fehler aufgetreten ist, im System-Register CR2 ab (s. Abschnitt 4.4). Zusätzlich legt er einen Fehlercode mit dem in Bild 4.8-5 dargestellten Format auf den Stack.

4.8.5 Fallstudie: Fehlercode des 80386 bei einem Seitenfehler

Bei bestimmten Traps ist es für den Interrupt Handler wichtig, die genaue Unterbrechungsursache zu kennen. Bei einer prozedurorientierten Ausnahme-Behandlung legt der Prozessor in diesen Fällen einen speziellen Fehlercode (*error code*), der oben bereits erwähnt wurde, auf den Stack des Trap-Behandlungsprogramms (s. Bild 4.8-4). Wird jedoch ein neuer Prozeß initialisiert, d.h. wird die Ausnahme-Situation prozeßorientiert behandelt, dann wird der Fehlercode auf den Stack des neuen Prozesses abgelegt.

Bei einem Seitenfehler *(trap 14)* wird auf den Stack ein Fehlercode mit dem im Bild 4.8-5 dargestellten Format abgelegt.

Bild 4.8-5. Format des Fehlercodes bei einem Seitenfehler

Die einzelnen Bits haben bei diesem Fehler die folgende Bedeutung:

P Das P-Bit unterscheidet, ob die Ursache des Seitenfehlers eine nicht im Hauptspeicher vorhandene Seite (P=0) oder eine Verletzung des Schutzkonzepts (P=1, *protection violation*) ist.

W/R Das W/R-Bit informiert darüber, ob der Fehler beim Schreiben oder beim Lesen der Seite auftrat.

U/S Das U/S-Bit zeigt an, ob der Seitenzugriff im Benutzer-Modus (*user mode*) oder im Betriebssystem-Modus (*supervisor mode*) erfolgte.

Anhand dieses Fehlercodes kann das Interrupt-Behandlungsprogramm also erkennen, ob es aufgrund einer im Hauptspeicher fehlenden Seite oder wegen einer Verletzung der Schutzmechanismen aktiv werden muß.